


For Reference

NOT TO BE TAKEN FROM THIS ROOM

Ex LIBRIS
UNIVERSITATIS
ALBERTAEASIS





Digitized by the Internet Archive
in 2021 with funding from
University of Alberta Libraries

https://archive.org/details/Thomas1972_0

THE UNIVERSITY OF ALBERTA

ALGOL 68 - CONSIDERATIONS FOR A ONE-PASS
IMPLEMENTATION

by



LLOYD KEITH THOMAS

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE
OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

FALL, 1972

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled "ALGOL 68 - Considerations for a One-Pass Implementation" submitted by Lloyd Keith Thomas in partial fulfilment of the requirements for the degree of Master of Science.

ABSTRACT

This thesis represents the results of an investigation into the implementation of a "one-pass compiler" for a modified version of the programming language ALGOL 68. Modifications made to the original language consist of additions and alterations that are considered to enhance the language from both a user's and an implementor's point of view. These additions and alterations are discussed, with examples, and a syntax for the modified version of ALGOL 68 is presented. Implementation dependencies and restrictions imposed to facilitate one-pass implementation are discussed, and techniques for some of the tasks which the compiler must perform are described. Special emphasis is placed on those tasks peculiar to a one-pass but not a "multi-pass" compiler.

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation and gratitude to Professor B.J. Mailloux, my supervisor, for his advice, guidance, and patience during the preparation of this thesis; to other members of the University of Alberta ALGOL 68 group, especially Mr. S.J. Wilmott and Mr. H.J. Boom, for many informative suggestions and discussions; to Miss P.J. Austin for her assistance in typing the thesis; and finally to *FMT and the IBM 360/67 at the University of Alberta for formatting and printing the thesis.

TABLE OF CONTENTS

Page

CHAPTER I: INTRODUCTION

1.1	Introduction	1
1.2	Notation and Syntax of the Report	2

CHAPTER II: CONSIDERATION OF MODIFICATIONS TO ALGOL 68

2.1	Introduction	5
2.2	Basic Tokens and General Constructions	8
2.3	Identification and the Context Conditions	11
2.4	Denotations	16
2.5	Phrases	17
2.6	Unitary Declarations	21
2.7	Unitary Clauses	25
2.8	Extensions	29

CHAPTER III: A ONE-PASS COMPILER

3.1	Introduction	33
3.2	Objectives	36
3.3	Restrictions	38
3.4	Implementation Dependencies	45

CHAPTER IV: A ONE-PASS IMPLEMENTATION

4.1	Introduction	49
4.2	Lexical Analyzer	50
4.3	Syntax Analyzer	52
4.4	Table Structure	63
4.5	Coercions and Identification of Operators	70
4.6	Code Generation	72
4.7	Runtime Environment	77

TABLE OF CONTENTS (continued)

	<u>Page</u>
CHAPTER V: CONCLUSION	79
REFERENCES	82
APPENDIX A: A MODIFIED ALGOL 68 SYNTAX	87
APPENDIX B: A CONTEXT-FREE SYNTAX	106
APPENDIX C: A REPRESENTATION LANGUAGE	120

LIST OF ILLUSTRATIONS

<u>Figure</u>	<u>Page</u>
3.1. Logical Parts of a Compiler	36

CHAPTER I

INTRODUCTION

1.1 Introduction

The Report on the Algorithmic Language ALGOL 68 [1] (hereinafter referred to as the Report) defines a new programming language intended to be the official successor to ALGOL 60 [2]. In comparison with ALGOL 60, ALGOL 68 is more powerful and has wider applicability. New data types ("modes") as well as new language constructions and operations are introduced. With this greater complexity, a more complicated descriptive technique is used. The description of ALGOL 68 consists of a syntactic part, formulated using a "two-level" syntax, and a semantic and pragmatic part, formulated in the English language.

Since the publication of the Report, many proposals concerning ALGOL 68 have been made. It is the first objective of this thesis to comment on ALGOL 68 and some of the proposals for additions and alterations. The second objective is to describe a proposed "one-pass compiler" for a large subset of ALGOL 68. Comments on additions and

alterations are made in the light of experience gained while designing the one-pass implementation.

The thesis presupposes considerable familiarity with ALGOL 68. The first occurrence of any technical term or notion from the Report will appear within quotation marks, and where necessary, will be followed by a reference to the particular section of the Report in which the definition may be found. Short descriptions and examples are given frequently to illustrate and clarify concepts being discussed.

Chapter II of this thesis is concerned with commenting on proposed additions and alterations to the language and developing a modified syntax for ALGOL 68. This syntax appears as Appendix A. Chapter III investigates general aspects of the compiler and discusses restrictions imposed to facilitate one-pass implementation. A description of some of the modules of which the compiler is comprised appears in Chapter IV, along with discussion of some of the major implementation difficulties. Finally Chapter V presents conclusions gained from the investigation.

1.2 Notation and Syntax of the Report

Unless otherwise noted, terminology and notation used is the same as that of the Report. References to specific parts of the Report appear between brackets and are given as

R followed by the section numbers and then, if necessary, a letter indicating the section or paragraph to which reference is being made. For example, [R1.3] or [R3.0.1.b].

ALGOL 68 is defined in three stages; the "strict language", the "extended language", and the "representation language" [R1.1]. The syntax of the strict language is a set of "production rules" for "notions". Sets of rules for the language are combined into single rules with the aid of "metanotions", the production rules for which are given by the "metaproduction rules" [R1.2]. A particular production rule is obtained from a rule of the Report by replacing all occurrences of a metanotion contained in the rule with the same terminal production of the particular metanotion. This is repeated for all metanotions throughout the rule. For example,

'actual real field letter t digit one declarator: virtual
real declarer, real field letter t digit one selector.'

is a particular production rule obtained from

'VICTAL NCNSTCWED field TAG declarator: virtual NCNSTOWED
declarer, NCNSTCWED field TAG selector.'

VICTAL, NCNSTCWED, and TAG are metanotions and have been replaced throughout by the terminal productions "actual", "real", and "letter t digit one", respectively.

Note that when reading rules of syntax ":" may be read as "may be a", "," as "followed by a", and ";" as "or a".

Detailed consideration will be given to the extended language and the representation language in Chapter II. The representation language used for examples given in this thesis is similar to that used in the Report.

CHAPTER II

CONSIDERATION OF MODIFICATIONS TO ALGOL 68

2.1 Introduction

The purpose of this Chapter is to discuss and comment on some proposals for additions and alterations to ALGOL 68. A syntax incorporating those additions and alterations thought to enhance the language appears as Appendix A of this thesis.

The following quotation is extracted from a formal resolution made at a recent meeting of Working Group 2.1 (ALGOL)¹ in Novosibirsk.

"... WG2.1 [Working Group 2.1] - in close collaboration with many implementors and potential users - is considering several proposals for correction, revision and also expansion of the language defined. These proposals may be classified in order of increasing degree of departure from the present language as follows:

1. Corrections and clarifications to the text of the Report without change to the body of the language, but with correction of

¹Working Group 2.1 (ALGOL), directed by Technical Committee 2 (TC2) of the International Federation for Information Processing (IFIP), commissioned and guided the writing of the Report on ALGOL 68.

those things which were obviously in error.

2. Changes of a notational nature within the existing conceptual framework of the language, in order to ease implementation and improve clarity.

3. Generalization of existing concepts and introduction of some new ones, so as to increase the power of the language and open possibilities for future enhancements.

4. New ideas, not ripe for inclusion in AIGCI 68. ..." [3]

These categories are referred to during discussion of certain proposals within this Chapter. No proposals clearly belonging to category four have been included in the modified syntax.

During the course of this investigation, many of the proposals for additions and alterations to ALGOL 68 were studied. While some of these proposals were adopted because they were considered to improve the language, others were rejected because they belonged to category four, for example [4-LQ215, 5-LQ201], or because they were thought to be unsuitable, from either a user's or an implementor's point of view, for incorporation into the language. Proposals considered within this thesis are referred as near to their source as possible from available documents and literature. Proposals not referenced may be considered to arise from this investigation (though perhaps simultaneously with similar proposals arising elsewhere).

Attached to most sections of syntax in the Report are semantic descriptions. With the syntactic modifications that are incorporated, corresponding modifications to the semantics are usually necessary. There are also some semantic changes belonging to category one that should be made, for example [6-proposal 8]. However, except for occasional mention of how the semantics might be modified, these semantic changes will not be further discussed in this thesis. Also proposals concerning "transput" [R5.5.1.aa] and associated matters [R10.5] are not considered in this investigation.

One of the major criticisms directed at the Report concerns the manner of presentation. In the preparation of this thesis the Report has been used as the defining document while the companion volume [7] has been used as a primer. In this manner, anticipation of the meaning of some sections of the Report was achieved and the task of comprehension relieved.

The first three chapters of the Report are introductory chapters; Chapter 0 is the "Introduction", Chapter 1 describes the "Language and Metalanguage", and Chapter 2 defines relationships between "The Computer and the Program".

The syntax of the metalanguage has been modified somewhat as other syntactic changes have been introduced. In

Chapter 2 a "particular-program" [R2.1.d] becomes a "serial-clause" [R6.1] enclosed between begin and end or between parentheses as suggested in [8-proposal 9].

The topics discussed in the remainder of this Chapter are presented in the same order and with the same titles as the Chapters of the Report, commencing at Chapter 3. Note that the syntax changes associated with some proposals may extend over Chapters of the Report other than the one currently being discussed.

2.2 Basic Tokens and General Constructions

Chapter 3 of the Report introduces the concept of a representation language, strongly suggesting rather than explicitly prescribing the representation(s) of symbols. The representation language for the proposed one-pass implementation to be discussed in this thesis is given in Appendix C.

Several modifications have been made to the syntax of Chapter 3 and to representation choices. It should be noted that "indications" [R4.2] have been kept separate from non-redefinable representations and that there is only one representation per indication. In the present Report for example, not and ~ are given as representations of the "not-symbol". This means that if the following "operation-

declaration" [R7.5] is made;

```
op not = (bool b) bool: true
```

then the meaning of the "operator" [R4.3] \neg is altered as well and this could be somewhat confusing to the user. Under the proposed modification, not and \neg will initially "possess" [R2.2.2.d] the same "routine" [R2.2.2.f, R2.2.3.4] but after the above declaration, \neg will remain as declared in the "standard-prelude" [R2.1.b, R10] (the declaration of standard "mode-identifiers" [R4.1], indications, operators, and "procedures" [R6.0.1.f]). This is easier to implement as no links need be kept between the representations \neg and not. For ease of writing operation-declarations in which the same routine is possessed by different "adic-indications" [R4.2.1.g], a proposal for "operator-bases" [6-proposal 30b] might be adopted though this has not been included in the modified syntax. Then the following would be possible;

```
op not = (bool b) bool: (b|false|true);
```

```
op  $\neg$  = (not); .
```

Full length representations of symbols that were abbreviated within the Report have been introduced. For example, procedure and proc are both representations of the "procedure-symbol". This is done at the request of some potential users [9-and others]. In the case of the "priority-symbol", a shortened representation is added.

There are other representations that have been added

in accordance with current proposals [10, 4-1Q207A] and with proposals to be discussed later in this Chapter. Since these changes involve representations they are all of category two. The "lexical analyzer" to be described in Chapter IV will be constructed so that it will be a relatively simple process to alter and/or add representations.

An objection to this representation language is the chosen "stropping" convention (see Appendix C). Many potential users have expressed annoyance at having to type an apostrophe before certain sequences of letters and digits. Thus in the design of the compiler, care has been taken not to make special use of the apostrophe so that a "reserved-word" representation language may be investigated. The reserved words would have blanks or special characters (characters other than letters or digits) as delimiters. Currently, there are five representations that might be confused with standard "identifiers" or "TAGs" [R4.1] in a reserved-word implementation. "in" and "out" possess routines defined in the standard-prelude while in and out are respective representations of the "in-symbol" and the "out-symbol". "re" and "im" are "field-selectors" [R7.1.1.i] defined in the standard-prelude while re and im are respective representations of the "real-part-of-symbol" and the "imaginary-part-of-symbol". Finally, "letter e letter x letter i letter t" [R2.1.e] might cause confusion with exit, a representation of the "completion-symbol".

These cases would have to be resolved if a reserved-word implementation was being considered.

2.3 Identification and the Context Conditions

Chapter 4 of the Report introduces identifiers, indications, and operators, describing methods for their identification and context conditions which a program must satisfy to be a "proper" program. "Applied occurrences" of identifiers (indications and operators) identify "defining occurrences" of identifiers (indications and operators) found by methods described in this Chapter of the Report. The identification processes make use of "ranges" [R4.1.1.e] which have much in common with what in some other programming languages are known as "blocks". In particular a range defines the "scope" [R2.2.4.2] of the "names" [R2.2.2.1, R2.2.3.5] created by declarations within it.

Because it would require a special check to test whether certain "mode-indications" [R4.2.1.b] could be redefined or not, it was decided to allow the primitive mode-indications, int, real, bool, char, and format to be redefined. Thus these five mentioned indications become "mode-standards" [R4.2.1.c].

The "short-symbol" is introduced and syntax changes are made so that as well as being able to have "long modes"

[R2.2.4.1], for example long real, it is possible to have "short modes", for example short int [11-proposal 1]. This gives an implementor the choice of internal representations for the mode-standards, int, real, bits, and bytes, while not restricting him from accessing shorter lengths should he not initially select the shortest length of internal representation on the physical machine. Note that it is now allowable to have any number of short-symbols or any number of "long-symbols" commencing an adic-indication. This unifies the implementation approach to adic-indications.

Some changes are incorporated for "dyadic-indications" [R4.2.1.d] and operator identification. Consider the following example;

```

priority + = 6;
op + = (int a,b) int: skip;
2+3; .

```

The first occurrence of + is an indication-defining occurrence. The second occurrence is an operator-defining and an indication-applied occurrence. The third is an operator-applied and an indication-applied occurrence. At present the operator-defining occurrence, as an indication-applied occurrence, must identify an indication-defining occurrence. It is proposed that in operation-declarations, no priority be associated with the operator-defining occurrences. Priorities are associated only with operator-applied occurrences (as indication-applied occurrences).

Thus the second + above would be an operator-defining occurrence but not an indication-applied occurrence. Necessary syntax changes have been incorporated and the corresponding alterations to the semantics, and to the process of operator identification, are not difficult. With the proposed changes, the first occurrence of + above is a terminal production of "priority-FIVE-dyadic-indication", the second a terminal production of "PRAM-dyadic-indication" and the third a terminal production of "PRAM-priority-FIVE-dyadic-indication". Examples such as the following are now permitted;

```

priority + = 6;
begin op min = (int a,b) int: (a<b|a|b);
    if ccndition then priority min = 5;
        print (3+2min4)
    else priority min = 7;
        print (3+2min4) fi; ,

```

where "condition" is of "boolean" mode. In the present language, this would violate a context condition [R4.4.1.c] and this context condition, because of the incorporation of the syntax changes, has been dropped. If "condition" in the above example is true, "4" will be printed, if false "5" will be printed. This proposal originated as a by-product of another proposal [4-LQ215]. If included independently, then a change to the definition of "protection" [R6.0.2.d] is necessary. Protection is the mechanism which allows

unhampered definitions of identifiers, indications and operators within ranges and permits a meaningful "call" [R8.6.2] within a range, of a procedure declared outside it. This is because in the semantics describing a call the actual procedure body is copied in place of the call. This is not in fact how calls would ever be implemented and it would seem better to eliminate the concept of protection, replacing it with semantics more closely paralleling implementation techniques. Nevertheless with the above scheme it is possible to redefine protection to allow the copying of procedures but there will be cases where, after protection, the identification process will still have to be carried out for dyadic-indications in order to ascertain their priorities.

In ALGOL 68, operators are identified not only by their indications but also by the modes of their "operands" [R8.4.1.c]. Currently, a programmer is not permitted to define two operators whose operands possess modes that might cause confusion. Such operands are said to have "loosely related" modes [R4.4.2.c]. It is proposed to replace the loosely related condition with the condition that a program is not proper if there exists an operator in a "formula" [R8.4] which identifies two operator-defining occurrences [4-LQ205A]. This resolves some ambiguities [12] and is not as restrictive as the current language. Under this new

proposal, operators with operands whose modes are loosely related will be permitted, unless an applied occurrence of one of the operators actually does lead to ambiguity. An increase in overhead may be expected in cases where there are several operators with the same indication as the applied occurrence declared within the same "reach" [R4.4.2.a] (a range excluding all ranges contained within it). All defining occurrences in this reach will have to be checked; the search may not terminate when the first applicable operator is found. However, when checking operators whose defining occurrences are in the standard-prelude, the search may terminate on finding the first applicable operator, because applied occurrences of operators declared within the standard-prelude are known not to lead to ambiguities. With the incorporation of this double identification context condition, there may be programs written in a subset of ALGOL 68 that are not proper programs in revised ALGOL 68. This is because ambiguities may arise due to the effect of certain features of ALGOL 68 not included in the subset. With the definition of "sublanguage" as in the Report [R2.3.c] this would mean that the subset was not a sublanguage.

Finally in this Chapter, another condition for "shielding" [R4.4.4.a] (avoiding the description of modes which require an infinite amount of storage) is added in that a mode-indication will be shielded if it is contained

in an "actual-bound" of a "declarer" [R7.1]. This will allow, as pointed out in [8-proposal 5], the following;

```
int i:=10; [1:10] ref [ ] real a;
mode m = [1:(i-:=1;
           if i>0 then int j = i;
           a[j]:=heap m fi; 6) ] real; .
```

This is equivalent to the following example;

```
int i:=10; [1:10] ref [ ] real a;
proc b = (int j) void: a[j]:=heap m;
mode m = [1:(i-:=1; if i>0 then b(i) fi; 6) ] real; .
```

This example is currently proper until the elaboration of the call when, due to textual replacement, context conditions become violated. No extra implementation problems are created, as the elaboration of an "actual-declarer" will involve the execution of a corresponding subroutine and this subroutine may be recursive.

2.4 Denotations

Known in other languages as "literals" or "constants", denotations are terminal productions of notions whose value is independent of the "elaboration" [R6.0.2] of the program. As published, the Report includes an unsatisfactory method of constructing "bits-denotations" [R5.2]. Essentially a bits-denotation is defined as being a sequence of "flip-symbols" and/or "flop-symbols". Actual representations of

bits-denotations using these symbols are unwieldy, for example, '1'0'1'1 or '1011. A proposal for "radix-denotations" [13-proposal 7], allowing more general bits-denotations with radices 2, 4, 8 or 16, is adopted; for example, 2r1010, 4r1231, 8r7605, 16r5a9f. The letters a through f are used for the hexadecimal "digits" 10 through 15. With the acceptance of this proposal the flip-symbol and the flop-symbol are no longer required and minor changes must be made to formatted transput with the introduction of "bits-patterns" [R5.5].

In ALGOL 68, routines and formats are recognized to be data objects and hence there exist "routine-denotations" [R5.4] and "format-denotations" [R5.5]. The syntax for routine-denotations is altered according to proposals to be discussed in sections six and seven of this Chapter.

With the introduction of short modes as mentioned in the previous section, "short-integral-denotations", "short-real-denotations", and "short-bits-denotations" are permitted.

2.5 Phrases

Phrases are "declarations" or "clauses". These are the bricks which make up a serial-clause and thus a particular-program. Three types of clause defined in Chapter 6 of the Report are the "collateral-clause" [R6.2], the "closed-

clause" [R6.3], and the "conditional-clause" [R6.4]. "Case-clauses" and "conformity- case-clauses" are introduced in the Report by means of "extensions" [R9.4]. However in the syntax presented in Appendix A these case-clauses are unified with the conditional-clause, as formulated in [14]. All extensions concerning these clauses are included in the syntax as is the proposal to require the matching of symbols in conditional-, case-, and conformity-case-clauses. For example, in a particular conditional-clause, it is permissible to use symbols from if, then, else, elsif, and fi, or from (, |, |:, and), but not from both. This required matching has been applied to other constructions in the syntax, for example, the "quote-symbols" delimiting "character-denotations" [R5.1.4] and "string-denotations" [R5.3], the "sub-symbol" and "bus-symbol" of "slices" [R8.6.1], etc.

In ALGCL 68 it is possible to have a mode "united from" [R4.4.3.a] other modes. For example, in

```
mode m = unicon(int, real);
```

m is united from the modes int and real. It is the purpose of "conformity-relations" [R8.3.2] to enable the programmer to find out the current mode of an "instance" [R2.2.1] of a "value" [R2.2.3] if the context permits this mode to be one of a number of given modes. With the introduction of the conformity-case-clause, which performs the same function, it

would seem that the conformity-relation is no longer necessary and it has been omitted from the syntax of Appendix A. However, perhaps after more experience of programming in ALGOL 68, it may be found that a need does exist in the language for conformity-relations.

The form of the proposed conformity-case-clause is illustrated by the following example;

```
casec unionintboolreal in (int): skip,
                                (bool b): skip
                                out skip cesac; .
```

Note that the following example is permitted;

```
casec unionintreal in (bool b): skip,
                                (char c): skip
                                out skip cesac;
```

in which case the "out-clause" is always taken (exactly as in if false then skip else skip fi). "unionintboolreal" and "unionintreal" are identifiers with obvious modes. The "(int):", "(bool b):", and "(char c):" in the above examples are called "specifications". A required context condition for the introduction of conformity-case-clauses is that no two specifications in the "conformity-units" of the "in-clause" of a conformity-case-clause may specify "related" [R4.4.3.b] modes. For example, in

```
casec unionintboolreal in (union(int, real)): skip,
                                (int): skip cesac;
```

if "unionintboolreal" possesses a value of integral mode

then it is not clear which alternative to take. The above context condition rules cut situations such as this because union(int, real) and int are related modes. Note the optional identifier in the specifications. If the instance of the value is to be used within the alternative, then an identifier, which is then made to possess that instance, may be included; if not, it may be omitted.

No change is made to the ranges of conditional- or case-clauses [15-proposal 18]. Thus in

```
if string x; read(x); x[1]≠"*" then string+:=x fi;
```

the "x" in the in-clause (the "then-clause") will not identify the "x" declared in the "conditional". The alternative is to write

```
begin string x;
```

```
if read(x); x[1]≠"*" then string+:=x fi end; .
```

The arguments for the proposed ranges do not seem strong enough to warrant changing the present situation. With the introduction of specifications within conformity-case-clauses, many of the examples using the proposed ranges are accounted for.

In Chapter 6 of the Report, "row-displays" and "structure-displays" [R6.0.1] are introduced. For example, in

```
[1:3] real x,y;
```

```
x:=(1,2,3);
```



```
y:=x+(2,3.14,4.5);
```

"(1,2,3)" and "(2,3.14,4.5)" are row-displays and "x" and "y" are of mode "reference-to-row-of-real". In

```
struct(int i, real x, char c) z;
```

```
z:=(1,2.3,"c");
```

"(1,2.3,\"c\")" is a "structure-display" and "z" is of mode "reference-to-structured-with-integral-field-letter-i-and-real-field-letter-x-and-character-field-letter-c".

At present it is permitted to have row-displays but not structure-displays as operands within formulas. With the new context condition for the identification of operators it is possible to allow structure-displays as operands because any ambiguities occurring will be found at the time of operator identification. However, the time taken to identify operators may be substantially increased if this is permitted. The syntactic changes required to allow structure-displays as operands are introduced in the syntax of Appendix A. Because of the practical advantages of having structure-displays as operands, it is planned to compare efficiency of operator identification between the cases when this is permitted and when it is not. Further discussion on displays appears in section seven of this Chapter.

2.6 Unitary Declarations

Unitary declarations are of four types. "Mode-

declarations" [R7.2] provide the indication-defining occurrences of mode-indications, "priority-declarations" [R7.3] provide the indication-defining occurrences of dyadic-indications, "identity-declarations" [R7.4] provide the defining occurrences of mode-identifiers, and operation-declarations provide the operator-defining occurrences of operators.

When specifying modes in ALGOL 68, declarers are used. Declarers are of three types: "actual", "formal", and "virtual" [R7.1], depending on the kind of "bounds" which are permitted. An actual-declarer describes an instance of a value of the corresponding mode while a virtual-declarer describes the corresponding mode only. Formal-declarers currently have the greatest freedom and can be used to check the compatibility of bounds between formal-declarers in "formal-parameters" [R5.4.1.e] and actual-declarers in "actual-parameters" [R7.4.1.b]. However, apart from specifying the "flexibility" of the bounds in "multiple values" [R2.2.3] (spoken of as "arrays" in some other languages) possessed by the corresponding actual-parameters, formal-declarers do not yield very much. Their use for checking compatibility of bounds is limited and a burden to the implementor. Operators lwb and upb are defined in the standard-prelude and these may be used to check bounds, if desired. It is therefore proposed to have only actual- and virtual-declarers [5-proposal 17]. However, the

specification of flexibility remains to be considered. It is proposed to specify flexibility in the mode, for example "flexible-row-of-real" specified by flex [] real, thus making flexibility a property of the whole multiple value, not of just one dimension. The either case (bounds which may be "fixed" or flexible), if needed, may be represented by union([] real, flex [] real). The modifications required for "flexible-row-of-" modes are difficult to formulate and represent some quite extensive changes to the Report (it would be a category three proposal). Attempts at a formulation allowing comparable use of flexibility to that described in the Report have not been successful and formal-declarers remain in the syntax of Appendix A. However, their function has been reduced to specifying the flexibility of bounds and apart from this they are the same as virtual-declarers. This proposal, together with another to forbid "go-on-symbols" in actual- and formal-parameter-packs [16] will prove useful in the "syntax analyzer" to be described in Chapter IV. These go-on-symbols were included to allow "side effects" as in the following example;

```
proc p = ([ 1:] real a; [1:upb a] real b) real: skip; .
```

However, they complicate the calling mechanism and if formal-declarers are modified as proposed, then there is no use for the go-on-symbol in formal-parameter-packs.

To improve "orthogonality" [R0.1.2], a new class of

modes, for example, `[] [] real`, is introduced to the language [13-proposal 5]. The declarer `[,] real` specifies "row-row-of-real" while `[] [] real` specifies "row-of-row-of-real". Thus it is now possible to have a "vector of vectors", for example, `[] string`. Given the declaration,

`[1:n,1:m] [1:1] real x`

it is possible to speak of `"x[2:3,5]"` or `"x[i,j]"` or `"x[i,j][k]"`, but not of `"x[i,j,k]"`.

To improve clarity for both the syntax analyzer and the user, the "void-symbol" is introduced for "virtual-void-declarers" [R7.1.1.z] [13-proposal 2]. For example, a routine-denotation of a routine which has an integral-mode parameter and which does not return a result may be written;

`(int a) void: skip .`

void, however, does not become a primitive mode such as int or real (new mode-standards). It may be that the syntax of the Report is simplified by the proposal to make void a mode [13-proposal 4], but the concept of a "void-mode" is somewhat confusing. In the Report, void has all the properties of an empty class of values and thus there would be no instance of a value of the mode void. It is not logical to declare

`void empty = skip`

as "empty" would then possess a value which is a member of the empty class of values, and this is contradictory. Since the class of modes denoted by void would be a subset of all

other classes of modes then it would not be logical to distinguish the modes union(int, real) and union(int, real, void). If void is considered to be a structure with no "fields" [R2.2.2.k] then there will exist instances of the mode void and these will occupy zero storage. Implementation problems then arise, for example, consider

```
[1:2] void a; a[1]:=a[2]; .
```

Normally the space occupied by an element of a multiple value will be equal to the "stride" [R2.2.3.3.b] of its last dimension. In order to be consistent with this, the "row-of-void", whose elements occupy zero storage, should have a stride of zero. Thus a[1] and a[2] will refer to the same address and the value of the "identity-relation" [R8.3.3] a[1]:=a[2] will be true, unlike similar language constructs. It would seem that void is only "virtually" a mode and thus it remains a virtual-void-declarer.

2.7 Unitary Clauses

Unitary-clauses (often abbreviated as "units") are the entities in the language which actually get things done. Unitary-clauses are either "coercends" [R8.2] or closed-, collateral-, or case-clauses. In the syntax of Appendix A there are five kinds of coercends. These are routine-denotations, "confrontations" [R8.3], formulas, "cohesions" [R8.5], and "bases" [R8.6]. Each of these coercends may have

any one of the eight basic "coercions" applied to it depending on its context, or syntactic position. These coercions are "dereferencing" [R8.2.1], "deproceduring" [R8.2.2], "proceduring" [R8.2.3], "uniting" [R8.2.4], "rowing" [R8.2.5], "widening" [R8.2.6], "hipping" [R8.2.7], and "voiding" [R8.2.8]. In the present Report there are five syntactic positions: "strong" positions, where all coercions may be applied; "firm" positions, where the first four coercions listed above may be applied; "weak" positions, where the first two may be applied; "soft" positions, where only deproceduring may be applied; and "EMPTY" positions, where no coercions may be applied. With the proposal to abolish conformity-relations, the only position in which coercions could not be applied (the right hand side of a conformity-relation) disappears and when conformity-case-clauses are introduced a new "meek" position replaces this "EMPTY" position. The meek position is essentially the same as weak except that it is possible to completely dereference the coerced.

Several proposals concerning coercion and belonging to category one have been incorporated into the syntax of Appendix A. These include the modifying of deproceduring as proposed in [17] to resolve an ambiguity; the allowing of

```
[ ] proc int pp = (i,j); pp;
```

in which "pp" is voided [18-proposal c]; and the hipping of "vacuums" [R8.2.6.1.b] to resolve an ambiguity, as proposed

in [8-proposal 1].

For clarity and ease of recognition, vacuums are now represented by an "open-symbol" followed by a "close-symbol". Thus vacuums appear as empty row-displays. Further, if the symbol row were required to precede row-displays, for example row(1,2,3), then row-displays of one element could be introduced. This would mean that a vacuum could be row(), that the rowing coercion could be eliminated, and that firm structure-displays could be permitted without affecting operator identification to as great a degree as described in section five of this Chapter. Correspondingly if array were required to precede declarers specifying "row-of-" modes then recognition of these would be simplified (see Chapter IV). At the same time, orthogonality would be improved if structures of zero or one fields and unions of zero or one modes were permitted. The first of these involves problems with a "structuring" coercion unless the symbol struct is required before structure-displays. If unions of zero or one modes were permitted then, perhaps union() would be equivalent to void, and union(a) would be equivalent to a. These suggestions have not been incorporated into the syntax of Appendix A.

With the proposal for "row-of-row-of-" modes, another type of rowing coercion is introduced. Rowing may now add another "row-" to a mode already beginning with "row",

inserting a new "quintuple" into the "descriptor" [R2.2.3.3], or it may add "row-of-" to any mode, constructing a new descriptor with one quintuple.

Though a proposal to remove proceduring from the language [4-IQ203A(7)] has met with some approval, proceduring may still be found in the modified syntax. Proceduring is an interesting concept and is probably no more confusing than some of the other coercions (or extensions, for that matter). The effects of proceduring on implementation will be discussed in Chapter IV. The following is an example in which proceduring is both elegant and useful.

```
cp andf = (bocl a, proc bool b) bool: (a|b|false);
```

```
bool:=bool1 andf bool2 andf bool3;
```

where "bocl", "bool1", "bool2", and "bool3" are of boolean mode. Without proceduring, the "assignment" [R8.3.1] would have to be written as follows;

```
bool:=bool1 andf (bool: bool2 andf (bool: bool3)) .
```

In order to make possible the construction of a sublanguage which does not contain any proceduring, a modified version of [13-proposal 1] is adopted. Routine denotations become units and are no longer packed between an open-symbol and a close-symbol. They do not become confrontations, because, in the following context they would then not be called;

...; void: skip;

"Casts" [R8.3.4] are made to look different from routine-denotations with the "cast-of-symbol" becoming ex or :: (if conformity-relations disappear). Also the "heap-symbol" is made mandatory in "global-generators" [R8.5.1] [19-proposal 2], the reasons for this being mainly didactic, with orthogonality being improved at the same time. It is then possible to allow assignments as "boundscripts" [R8.6.1.1.1] because boundscripts may now be units without causing ambiguity. For example, a[int:i] has a routine-denotation as a "subscript" [R8.6.1.1.i], a[heap int:i] has a global-generator as the lower bound of a "trimmer" [R8.6.1.1.f], and a[int ex i] has a cast as a subscript. Similar examples with assignments are a[i:=int:j], a[i:=heap int:j], a[i:=int ex j].

2.8 Extensions

Chapter 9 of the Report introduces extensions, which, when applied to a program in the strict language, produce a program in the extended language. Extensions, which mainly concern "comments", declarations, "repetitive statements", and case-clauses, cause ALGOL 68 programs to look more like programs in certain other languages. However, extensions seem out of place in a rigorously defined language such as ALGOL 68. Their method of description employs no formal

techniques and, consequently, automatic methods of recognizing a program in the extended language are impossible. If the features introduced by extensions are deemed necessary to be in the language then they should be formally defined as part of the strict language. Some features, for example "options" [R3.0.1.b], presently in the definition of the strict language are no more than extensions anyway. Thus all extensions were studied and only those considered necessary are included in the syntax of Appendix A.

Comments [R9.1] are included in the syntax by means of "tokyls" which are one syntactic level higher than "symbols", the terminals of the strict language. A tokyl may optionally be replaced by a comment-sequence followed by the particular symbol. Care has been taken so that comments do not appear within comments or string- or character-denotations.

Contracted conditional-clauses, case-clauses, and conformity-case-clauses [R9.4] have been unified with conditional-clauses as mentioned in section five of this Chapter.

A syntax for repetitive statements [R9.3] is incorporated so that a "repetitive-clause" becomes a production of unitary-clause, repetitive statements being specified by the extension as "strong-unitary-void-clauses".

The form of the repetitive-clause has been altered so as to allow the definition of a meaningful range incorporating the "control-identifier", the "while-clause" and the "do-clause". Also all constituent clauses become serial-clauses. For example;

```
from 4 by 2 to 100 for i while condition do skip; .
```

The extensions concerning bounds [R9.2.f] and sub- and bus- symbols [R9.2.g] are easily incorporated into the syntax while another extension [R9.2.d] has been obviated by the acceptance of a proposal concerning routine-denotations as in section seven of this Chapter.

The remaining extensions concern declarations and as demonstrated in [4-LQ203A(4)] and [20-proposal 2] cause some problems, especially since the order of applying extensions becomes important. If extensions had been formally defined it is doubtful that these complications would have arisen, and those extensions which cause confusion have not been included in the modified syntax. Extensions across a "becomes-symbol" are also excluded. For example,

```
loc proc (int) int := (int a) int: skip;
```

may not be contracted to

```
loc proc := (int a) int: skip; .
```

In cases such as this the left-hand side is a generator and not a formal parameter as in similar contractions. Generators cause the reservation of storage but this will

have to be postponed, if such extensions are allowed, until the contraction has been "uncontracted".

Many semantic modifications accompany the inclusion of extensions in the syntax. For example;

$$[1:n:=n+1] \underline{\text{int}} \ a, b;$$

is a contraction of

$$[1:n:=n+1] \underline{\text{int}} \ a, [1:n:=n+1] \underline{\text{int}} \ b;$$

using present extensions. Thus, the assignation in the contraction should be elaborated twice contrary to what a user may suppose. In such declarations actual-bounds should be elaborated only once and the semantics will have to reflect this. Perhaps a suitable method of semantic description would regard the whole declaration as a "structure", the actual-declarer of which is elaborated once only.

CHAPTER III

A ONE-PASS COMPILER

3.1 Introduction

ALGOL 68 is defined in terms of "actions" performed by a "hypothetical computer" [R2.2]. A model of this hypothetical computer, using a physical machine, is an "implementation" of ALGOL 68. A sublanguage of ALGOL 68 is a language whose particular-programs are particular-programs of ALGOL 68 and have the same meaning. In an implementation, the particular-program may be translated into a program in the machine language of the physical machine. This translation is best performed by the computer itself using a special program called a "compiler". The particular-program or "source program" is written in the "source language", ALGOL 68, and is compiled into the "object program", written in the "object language" or "object code" of the physical machine. The translation of the source program into the object program occurs at "compile-time", and the execution of the object program at "runtime".

A compiler must perform an "analysis" of the source

program and then a "synthesis" of the object program. Schematically, the logical parts of a compiler are shown in Figure 3.1. This diagram represents a logical connection, rather than a time connection of the compiler parts. The processes can be performed in the order depicted by Figure 3.1, or they could be performed in a parallel, interlocked manner. A compiler which performs these processes in the former fashion is a "multi-pass" compiler; one that performs them in the latter fashion is a "one-pass" compiler. In a one-pass compiler, the source text is processed once, and executable object code produced immediately. Actually, very few compilers are truly one-pass. For example, the place in the program to which control is to be transferred by a jump instruction (in ALGOL 68, go to 1, for example) is not generally known when the jump instruction is encountered, unless the transfer point precedes the jump. Thus some sort of "back-tracking" is necessary.

Some languages are structured so that they cannot be translated in a reasonable fashion by a one-pass compiler. ALGOL 68 is, in fact, one of these languages. Restrictions to the language are necessary and these will be discussed in section three of this Chapter. Also in ALGOL 68, there are cases where it is necessary to scan ahead in the source text in order to resolve some indeterminations. These cases will be discussed in Chapter IV of this thesis. Thus the proposed

implementation is actually one-pass with some "cheating", in the form of back-tracking and looking-ahead.

In a one-pass compiler it is not necessary to store, for any length of time, intermediate object code that might be produced by a multi-pass compiler. Nor, in a one-pass compiler, should any functions of the compiler be duplicated. For example, in a multi-pass compiler, certain sections of the source text, or slight modifications thereof may be analyzed more than once, each time by a different pass.

In a multi-pass compiler, on the other hand, it is possible to produce more efficient object code for larger subsets of the language and presumably the language itself. Thus the choice of a one-pass or a multi-pass compiler depends largely on the objectives of the implementor.

For the historical developments and a discussion of the general techniques of compiler-writing, the reader is referred elsewhere. In particular, [22,23,24,25] were used in the course of this investigation.

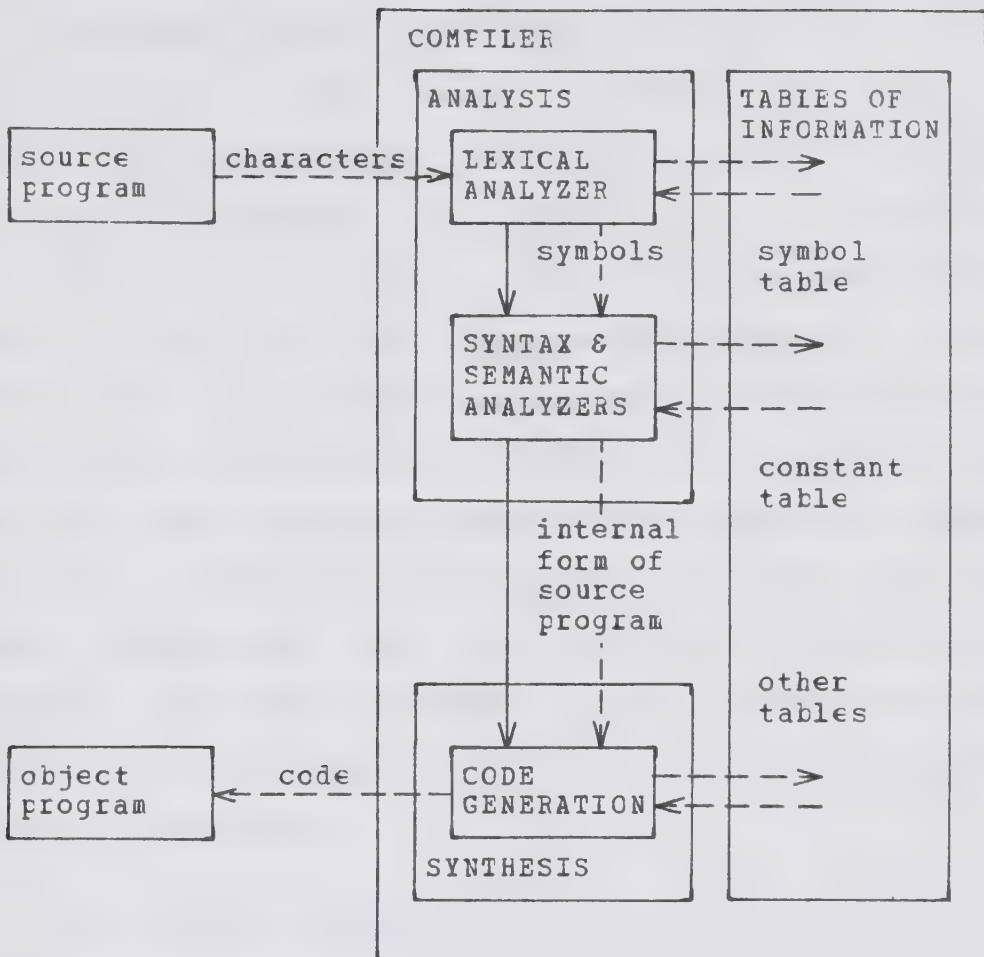


FIGURE 3.1. Logical Parts of a Compiler.

3.2 Objectives

The objectives of this implementation are as follows:

- (i) to implement a one-pass compiler for a large subset of ALGOL 68.
- (ii) to be modular in design.
- (iii) to provide facilities for good error diagnostics and runtime "debugging".

A one-pass compiler was chosen in order to provide fast turn-around (time taken to compile and execute a program) for normal, small-to-medium size ALGOL 68 programs, the type of programs one finds in an educational environment. It is planned that the whole system should reside in core, the only input/output being the source program itself and the input/output which the source program carries out. Only minimal local optimizations of object code will be made, producing relatively inefficient object programs in absolute machine language, which the system will execute immediately. Thus no expensive linkage-editing (combining of object programs) and/or loading (separate insertion of a relocatable object program into the physical machine) is necessary.

The modular design is employed so as to keep the various tasks of the compiler distinct and well-defined. It is proposed that some modules be included or excluded depending on the setting of particular program switches. In this way it will be possible to study the relative effects of some of the features of ALGOL 68 on both compile-time and runtime efficiency, and on the size of the compiler. Consequently, information concerning the development of sublanguages for ALGOL 68 [26] may be obtained.

Because of the possible use of the compiler in an

educational environment (or in any environment where programs are being developed and not just run), it is necessary to provide facilities for good error diagnostics and runtime debugging, all too often neglected in the design of compilers for high-level languages.

3.3 Restrictions

The following restrictions are made to ALGOL 68 in order to facilitate the one-pass implementation.

1. Add the following context condition [R4.4.1.d] concerning identification.

"d) No proper particular-program in the sublanguage contains an applied occurrence of a mode-identifier (indication-applied occurrence of a dyadic indication, operator-applied occurrence of an operator) which is textually before its defining occurrence."

Consider the following example;

```
begin real x;
```

```
    proc p = void: begin a x; ... end; ... .
```

In a strictly left-to-right scan of this text it is not known whether a x is a "monadic-formula" [R8.4.1.g] or a declaration of a local variable "x" until the defining occurrence of a is encountered. It is for this reason that Mailloux suggests in [27] a special pass to process mode-

indications, dyadic-indications and operators, based on their defining occurrences. With the identification condition above, a in the example could only be a mode-indication but a context condition to be introduced later will also exclude this case. The fact that this example is illegal will be reported at inspection of a in a x because no defining occurrence of a has yet been encountered.

The problem with mode-identifiers is obvious from the following example;

```
begin int i;
      begin proc p = void: begin i:=1 end;
      int i; ... .
```

Again, in a left-to-right scan, it is not known when inspecting the applied occurrence of "i" in i:=1 whether it identifies the defining occurrence in the first line or one that occurs later, as is the case here. It is for this reason that Mailloux suggests a further pass to process identifiers based on their defining-occurrences (now recognizable due to the proposed first pass) in identity-declarations (and in "labels" [R6.1.1.k] though these "label-identifiers" [R4.1.1.b] are treated separately here). The example is illegal in the sublanguage and this should be reported on inspecting the defining occurrence of "i" in int i on the third line. It is illegal because an applied occurrence has occurred textually before its defining occurrence but matters are further complicated because

another defining occurrence was applicable (during the left-to-right scan) when inspecting "i" in $i:=1$. Thus some record must be kept of applied occurrences of mode-identifiers, and for similar reasons, of operators and indications (including mode-indications) so that errors such as this may be reported. Mailloux points out that policing this item is likely to be almost as costly as keeping the first two passes and the unrestricted language. Methods of recording applied occurrences exist and one of these will be described in Chapter IV of this thesis.

With the introduction of the above identification condition, a modification of that presented in [27], the identification problems concerning mode-identifiers, operators, and dyadic-indications are resolved, requiring "things to be defined before their use". As noted by Mailloux, this is not a serious restriction in that "sensible" programmers "always" declare things before their use. However it is now more arduous to write mutually recursive operators and procedures. It is fortunate that ALGOL 68 leaves a loophole which enables this to be done despite the requirements of the above identification condition. For example;

```
op p = (int a) real: begin ...; g a; ... end;
```

```
op g = (int b) int: begin ...; p (b-:=1); ... end;
```

is now illegal, but may be rewritten as follows;


```

proc (int) int q1;
op p = (int a) real: begin ...; q1(a); ... end;
proc q = (int b) int: begin ...; p (b-:=1); ... end;
q1:=q; op (int) int g = q; .

```

The reason that mode-indications were not included in the above condition is that it is possible to have mutually recursive modes in unrestricted ALGOL 68, and no loophole such as the one employed above for operators exists to enable the alternative declaration of these mutually recursive modes. For example, the following mode-declarations cannot be rewritten to avoid recursive definition:

```

mode a = struct(ref b a1, ref c a2, int a3);
mode b = struct(ref c b1, ref a b2, real b3);
mode c = struct(ref a c1, ref b c2, bool c3); .

```

Thus another context condition is defined which allows recursive mode-declarations but requires definition before actual use.

2. Add the following declaration conditions [R4.4.4].

"d) If an indication-applied occurrence of a mode indication identifies an indication-defining occurrence of that mode-indication, then it must also 'ultimately' identify one or more indication-defining occurrences of mode-indications found by the following steps:

Step 1: Each mode-indication is said not to have been encountered; the given indication-applied occurrence is considered;

Step 2: The considered {indication-applied} occurrence and all mode-indications which are the same terminal production of 'MODE-mode-indication' are said to have been encountered; the indication-defining occurrence identified by the considered occurrence is said to be ultimately identified by the given occurrence, and is itself considered;

Step 3: If the constituent declarer of the mode-declaration, of which the considered {indication-defining} occurrence is a constituent contains one or more indication-applied occurrences of not yet encountered mode-indications (other than occurrences contained within a boundscript contained within that declarer), then each such {indication-applied} occurrence of each such mode-indication is considered in turn, and for each one Step 2 is taken.

e) No proper particular-program in the sublanguage contains a generator (a formal-parameter) whose constituent declarer contains a mode-indication (other than a mode-indication contained within a boundscript contained within that declarer) which ultimately identifies an indication-defining occurrence of a mode-indication which occurs later in the textual order than the given declarer."

This condition is taken almost exactly from [28] and is actually simpler than it appears. For example, in


```

struct a = (real p, ref b q);
union b = (bool, c);
mode c = proc (ref a) void;
a x;

```

the applied occurrence of a in the last line "ultimately" identifies the indication-defining occurrences of a, then real (in the standard-prelude) and b, then bool and c. In the identity-declaration a x (contracted from ref a x = loc a) a is a generator and since all indication-defining occurrences which it ultimately identifies textually precede it, the example is thus permitted. However,

```

union a = (real, ref b);
a x;
mode b = int;

```

would not be allowed since when the declaration a x is encountered a ultimately identifies real and b and no defining occurrence has yet been encountered for b.

It would not be in order to require that defining occurrences of label-identifiers precede applied occurrences, for then

```

... go to l; ... ;l: ... ;

```

would be illegal. In ALGOL 68 it is possible to omit the go to in a "jump" [R8.2.7.1.c] and this can cause problems for the one-pass compiler. For example, in


```

proc p = void: skip;
begin p;

    p: skip end;

```

it is not known whether the second occurrence of p is a jump or not. Thus go tos will be required in every jump. This may be more restrictive than necessary but it is less complicated to explain. For example, in

```

proc p = void: skip;
begin p: skip;

    p end;

```

there is no problem and the go to has been omitted. However the requirement of go to alleviates problems such as recognizing (11:12), by reducing the number of possibilities for this construction. The following example,

```

p: skip;

begin go to p;

    p: skip end;

```

is legal for label-identifiers unlike the corresponding example for mode-identifiers. The compiler must carry out back-tracking in order to resolve examples such as the latter. Thus the third restriction is as follows.

3. The syntax for a jump becomes [R8.2.7.1]

"c) MOID jump: go to symbol, label identifier."

4. The final restriction concerns parallel processing.

There is no parallel processing and no use is to be made of the fact that the user might write commas instead of semicolons, giving "collateral elaboration" [R2.2.5.a, R6.2.2]. The "parallel-symbol" will be ignored and if the "sema-symbol" is used, an error message will ensue.

3.4 Implementation Dependencies

When designing a compiler for ALGOL 68 it is necessary to choose the number of different lengths of integers and reals to be implemented. ALGOL 68 makes provision for an infinite number of lengths but to actually have an infinite number of lengths is, of course, impractical. It is then necessary to choose "internal" [R2.2.1] representations for integral and real modes so that long modes and the newly-added short modes may also have representations. As an example of this, consider a physical machine with "full-word" and "half-word" integer arithmetic and "full-word" and "double-word" real arithmetic. Then it would be possible to select a full-word to represent integers and a half-word to represent short integers. For reals, real and long real may correspond to full- and double-words respectively. Certain choices regarding hits and bytes must also be made.

In the standard-prelude of the Report there is a section entitled "Environment Enquiries" [R10.1] in which definitions are given for a number of identifiers and

operators that enable the programmer to ascertain certain features which are dependent on a particular implementation. The function of the identifiers is to supply information such as the number of different lengths of integers (the identifier "int lengths") and reals ("real lengths"), the number of widths of bits ("bits widths") and bytes ("bytes widths"), the largest integral value ("max int"), the largest long real value ("long max real"), the smallest real value that can be meaningfully added to or subtracted from one ("small real"), and so forth. With the introduction of short modes, it is likely that additional environment enquiries will be made available so that, for example, the number of short integer lengths and long integer lengths may be ascertained. Two operators, abs and repr are also defined such that abs "a", for example, will return the integral equivalent of the character "a", and repr 6, for example, will return that character "x", if it exists, such that abs "x" = 6. Further environment enquiries in the standard-prelude concern transport, for example, enquiries concerning "channels" [R10.5.1.1] and "external" [R2.2.1] representations [R10.5.2.1] (the number of decimal digits required to represent accurately the chosen internal representations). These features all depend on the particular machine on which ALGOL 68 is being implemented, as well as depending on the implementor himself.

The following example illustrates a problem. Let the number of lengths of real be 2, i.e. "real lengths = 2", corresponding to real and long real, say. Then what if the programmer writes

```
long long real x := long long max real; ?
```

In the proposed implementation, a warning will be given in such cases and the above assignation will be compiled as

```
long real x := long max real;
```

with "x" being considered to be of mode long long real, in complete accordance with the Report.

The number of longs able to precede a declarer depends largely on how modes are stored. For example, long long long real may be stored with a special counter in its description, which in this case would be set to 3 (for shorts this counter could be negative). The situation is similar for the number of refs and the number of rows (i.e. "row-"s or "row-of-"s) that may precede a mode. Note however, that runtime considerations will certainly restrict the number of rows. In the above case, where a counter is used, the maximum number of longs permitted would depend on the maximum value which the counter could assume. Other methods for storing modes exist [29] where the number of longs, shorts, refs, or rows might depend on table size rather than on the size of a counter. In the proposed implementation, the restrictions will not be severe. The number of longs or shorts will depend on the size of a counter, the number of

refs on table size, and the number of rows on runtime considerations. No syntactic changes reflecting these restrictions have been incorporated into the syntax of Appendix A.

It should be noted that the Report allows an infinite number of identifiers to be defined within the standard-prelude. For example, "long max int", "long long max int", "long long long max int", etc. There will have to be a special technique for storing and looking up such identifiers.

Two other possible implementation dependencies are the length of string-denotations and the maximum number of letters or digits permitted in identifiers and indications. In the proposed implementation, it is hoped that these limits will depend on table size (such entities must be stored) rather than on some feature of a particular physical machine.

CHAPTER IV

A ONE-PASS IMPLEMENTATION

4.1 Introduction

This Chapter proposes an outline for a one-pass compiler for the sublanguage of ALGOL 68 detailed in Chapter III of this thesis.

ALGOL 68 incorporates many concepts from other programming languages, and basic techniques developed for the compilation of these other languages are sometimes applicable or perhaps partly applicable to the design of a compiler for ALGOL 68. For instance, methods of handling structured values are known, at least partially, from list- and string-processing languages (for example, LISP [30] and SNOBOL [31]). Runtime organization of ALGOL 68 implementations employs stack techniques which, in principle, may be used for ALGOL 68. However other concepts introduced in ALGOL 68 make implementation difficult, and cause problems not previously encountered in the design of compilers for high-level programming languages.

The remainder of this Chapter discusses the major components of the compiler and system, giving techniques or references to techniques for some of the tasks which must be performed within these components. Particular emphasis is placed on those techniques that are peculiar to a one-pass but not a multi-pass compiler.

4.2 Lexical Analyzer

The lexical analyzer or "scanner" is that part of the compiler which scans the characters of the source program from left to right, constructing the complete symbols of the program, as well as composing denotations and identifiers. In general, the operations of lexical analysis are well-defined and standard finite-automaton techniques [23-Chapter 3, 32] will be used for the scanner of the proposed one-pass compiler.

More specifically, the lexical analyzer must perform the following functions:

1. Do the bookkeeping required to know where in the input stream the compiler is operating, fetching new source records and listing these if required. This part of the scanner may be somewhat implementation-dependent due to the input/output operations that are involved.
2. Ignore comments and insignificant blanks (those not in string-denotations) that occur in the source program.

3. Perform actions specified by "compiler directives" or perhaps as suggested in [R2.3.c] "pragmats". Pragmats enable the user to specify, for example, whether or not a listing of the source text is required, the column range of source records, optional features that the compiler provides, etc.
4. Compose symbols from the characters of the source text. These symbols will have representations according to the representation language given in Appendix C of this thesis.
5. Compose identifiers from sequences of letters and digits.
6. Compose and convert to internal form all denotations except routine- and format-denotations.

In the representation language of Appendix C, care has been taken to ensure that a sequence of characters may be uniquely partitioned into symbols. For example,

$$a+:=\neg:/?*b$$

may be partitioned uniquely as follows:

$$a \quad +:= \quad \neg:/ \quad ?* \quad b$$

where "a" and "b" are mode-identifiers, $+:=$ is a dyadic-operator and $\neg:/$ and $?*$ are monadic-operators. This unique partitioning is desirable because the number of possibilities for syntactic analysis is thereby reduced. The syntax given in Appendix C for "dyadic-indicants" and "monadic-indicants" [R4.2.1] which consist of special characters (characters other than letters or digits) are expressed using regular grammars [23-Chapter 3] to

facilitate construction of the lexical analyzer.

If a given representation represents more than one symbol then some complications arise. For example, `+`, `-`, `=`, `|`, `|:`, `in`, `out`, `:`, `(`, and `)` are such representations according to the representation language given in Appendix C. For `+`, `-`, and `=` the complications are minor and are easily resolved because of the limited contexts in which the corresponding symbols may appear. For instance, `+(-)` may appear as a representation of the "plus-symbol" ("minus-symbol") in an "exponent-part" of a "real-denotation" or in a "sign-frame" within a format-denotation. `+(-)` may also appear as a representation of the "add-symbol" ("subtract-symbol") within formulas, operation-declarations, or priority-declarations. These cases are easily distinguishable, as are the cases for `=` which will not be detailed here. For `|`, `|:`, `in` and `out` the complications are also minor and the immediately surrounding context will determine exactly which symbols these representations represent. Complications with `(` and `:` are more serious and these will be discussed in the next section of this Chapter.

4.3 Syntax Analyzer

The syntax analyzer or "parser" analyzes the source program, decomposing it into its constituent parts, and calling routines to perform specific functions dependent on

the type of construct currently being analyzed. The syntax analyzer forms the nucleus of the proposed one-pass compiler.

Most of the methods developed for the syntactic analysis of programs presume that the language in question is at least bounded-context [33]. ALGOL 68 does not fulfil this condition and is not amenable to the usual, context-free analysis methods [23-Chapters 4,5,6]. Research on the automatic analysis of ALGOL 68 programs, guided by the two-level syntax, is currently being carried out [34] but such techniques will not be used in the proposed implementation. The method of parsing selected for the proposed compiler is that of a "top-down recursive descent" analyzer based on a context-free syntax which describes a superset of ALGOL 68 and which appears as Appendix B of this thesis. A context-free grammar for the language described by the two-level syntax of Appendix A could not be generated automatically from the syntax of Appendix A, since the two-level syntax generates an infinite number of context-free production rules. It is for this reason that the syntax of Appendix B describes a superset of the language described by the syntax of Appendix A. No problems arise however, as improper programs which will be parsed as correct programs in the superset, will be detected by routines called by the syntax analyzer. During the development of the context-free syntax,

much use was made of [35].

The top-down method of analysis known as recursive descent [23-p. 97] is described as follows: for a given nonterminal U , there is a recursive procedure which parses phrases for U . The procedure is told where in the source text to begin looking for a phrase for U , and hence the method is "goal-oriented" or "predictive". The procedure finds the phrase by comparing the text at the point indicated with right parts of production rules for U , calling other procedures to recognize subgoals when necessary. Thus the subject language syntax is built into the analysis program. Some advantages of this method are that rules can be rearranged to fit the needs of the procedures and to improve efficiency of the parsing process, and that other routines for further syntactic analysis, for checking context conditions, and for code generation may be inserted anywhere within a procedure and not just at the end when the phrase is detected. Two disadvantages with the recursive descent method of analysis are that more programming and debugging is required and that modifications to the syntax are harder to incorporate than if other methods of analysis were used. However, neither of these disadvantages is thought to be severe. In particular, in an effort to reduce compile-time, the method of recursive descent was selected and methods using Floyd-Evans Production Language [36, 37] or modifications thereof, which

are employed in some other implementations of ALGOL 68 [38, 39], were rejected.

It should be noted that the syntax analyzer is not to be constructed directly from the syntax of Appendix B but rather from a contracted syntax in which rules are ordered to obtain a minimal amount of back-tracking.

The syntax of Appendix B contains two intrinsic ambiguities. The first involves certain constructions which may be parsed as either slices or calls; for example, $a(i,j)$. However, in Chapter III of this thesis, a restriction was imposed to ensure that the defining occurrence of a mode-identifier would precede any applied occurrence. In an example such as $a(i,j)$ then, the defining occurrence of "a" should have been encountered and whether the given construction is a slice or a call may be determined by inspecting the mode of "a". If no defining occurrence of "a" exists, then the program containing the construction is not a proper particular-program in the sub-language and an error message will result. The second intrinsic ambiguity in the context-free syntax involves "indicants" [R4.2.1], monadic-indicants, and dyadic-indicants. For instance, \underline{a} may be a terminal production of any of these and consequently $\underline{a} \ a$ and $(1:1) \ \underline{a} \ a$ each allow two interpretations. In the case that \underline{a} is an indicant, the two constructions above are identity-declarations. If \underline{a} is a

monadic-indicant in the first construction or a dyadic-indicant in the second construction then the constructions are formulas. This ambiguity is resolved by way of restrictions imposed in Chapter III, by requiring mode-indications, monadic-operators, and dyadic-indications (dyadic-operators) to be declared before use.

Major complications in the syntactic analysis are caused by the indeterminations that exist concerning left parentheses and right parentheses (less importantly, since a right parenthesis must have a corresponding left parenthesis). In ALGOL 68 parentheses are "overloaded", that is, there are many constructions which parentheses may delimit. According to the syntax of Appendix A and the representation language of Appendix C the left parenthesis is either a representation of open-symbol or a representation of "brief-conditional-begin-", "brief-case-begin-", or "brief-conformity-case-begin-symbol". A left parenthesis encountered in a left-to-right scan of the source stream may fall into one of two categories as follows:

(i) A left parenthesis, the left context of which is sufficient to determine the construct which the left parenthesis precedes. For example, in each of struct(int i, real r) and struct s = (real a, b) the struct to the left of the left parenthesis determines the following construct as a

"FIELDS-declarator" [R7.1.f] [40].

(ii) All other left parentheses. The following are constructs (with examples) that a left parenthesis belonging to this category may begin:

vacuum	()
routine-denotation	(<u>a</u> a) <u>b</u> : <u>skip</u>
actual-declarer	(l1:l2) <u>a</u> a
virtual-declarer	(, :) <u>a</u> a
closed-clause	(<u>a</u> a; <u>skip</u>)
collateral-clause	(<u>skip</u> , <u>skip</u>)
conditional-clause	(bool <u>skip</u> <u>skip</u>)
case-clause	(int <u>skip</u> , <u>skip</u> <u>skip</u>)
conformity-case-clause	(union (<u>a</u> a): <u>skip</u> <u>skip</u>)

where a and b are mode-indications, and "l1", "l2", "a", "bool", "int", and "union" are identifiers. Note that the left parenthesis beginning a specification (in the new conformity-case-clause) belongs to category (i) as the specification is distinguishable by context to the left.

When analyzing the program, context to the left is taken into account by the mechanisms of recursive descent and so parentheses belonging to category (i) cause no complications. However, in general, an infinite context to the right may be required to distinguish the parentheses in category (ii). Two approaches to this problem will be discussed. First, it is possible to consider all interpretations of the construction in parallel, eliminating

cases as analysis of the source text proceeds. Finally when context to the right has eliminated all but one case, the effects of the incorrect interpretations must be cancelled. This approach is very difficult for a one-pass compiler and would require extensive back-tracking. The second approach, the one adopted in the proposed implementation, is to employ a look-ahead algorithm which will be initiated at left parentheses belonging to category (ii) and will divide the constructs into at least four classes as follows: (a) vacuums, (b) virtual-declarers, (c) routine-denotations, and (d) all other constructs beginning with left parentheses belonging to category (ii). Fortunately, it is not immediately necessary to distinguish all cases mentioned in category (ii). The particular construct that must be discerned is the routine-denotation. For example, consider a in (a a; skip) and (a a) b: skip. In the first case, a closed-clause, a a is an identity-declaration. In the second case, a routine-denotation, a a is a formal-parameter. These constructs require different actions by the compiler and thus the cases must be distinguished.

Note that if array was required before declarers specifying "row-of-" modes then virtual- and actual-declarers would not appear in category (ii) above and complications would be somewhat reduced.

The look-ahead algorithm is simple in principle and is

based primarily on the particular symbols that may appear between left and corresponding right parentheses, excluding any text surrounded by inner parentheses. The following are the particular symbols for the first three of the above four classes:

- (a) vacuums: empty (i.e. no symbols at all).
- (b) virtual-declarers: empty, ",", :, "mode-words", mode-indications, mode-identifiers, tags.
- (c) routine-denotations (i.e. formal-parameter-packs): ",", mode-words, mode-indications, mode-identifiers, tags, (,), [,], (/, /), :, flex, either (these latter three may appear only between (/ and /) or [and]).

Mode-words include the following symbols which are associated with declarers: proc, union, struct, ref, long, short, and void. For vacuums and virtual-declarers the look-ahead algorithm scans past the corresponding right parenthesis, to distinguish between cases such as () and ()real. As soon as a decision is reached the algorithm terminates. Extra information is utilized in this connection; for example, the occurrence of | or |: implies a conditional-, case-, or conformity-case-clause; the occurrence of ; implies a conditional-, case-, conformity-case-, or closed-clause. Note that this algorithm assumes the acceptance of modifications to formal-parameter-packs and formal-declarers as outlined in Chapter II and incorporated in the syntax of Appendix A of this thesis. The

new restrictions concerning formal-declarers imply that the algorithm should not usually have to look far to the right, even though technically speaking, the context to the right is still unbounded. The look-ahead algorithm will be attached to the syntax analyzer and lexical analyzer in such a way that lexical analysis will not be performed twice.

Following the application of the look-ahead algorithm, a construct beginning with a left parenthesis and belonging to class (d) as defined above may be a closed-, collateral-, conditional-, case-, or conformity-case-clause, or an actual-declarer. Except for one special case a construct such as this is processed from left to right with further discernment occurring as more source text is analyzed. The remaining special case is the locally ambiguous construction (l1: expression) where "l1" is an identifier and "expression" is a unit. Here the overloading of : causes problems. If : is the "label-symbol" then "l1" is a label-identifier and the construction is a closed-clause. If : is the "up-to-symbol" then the construction is part of an actual-declarer specifying a "row-of-" mode. Because of the definition-before-use restriction, if "l1" is not defined then it may only be a label-identifier. However, if it is defined, then it may be either a label-identifier or a mode-identifier and the construct may correspondingly be a closed-clause or part of an actual-declarer. Note that the

case where "l1" is a jump has been excluded in the considered sublanguage. The identifier that occurs between the open-symbol and the label- or up-to-symbol of this special case will be called, for the purposes of this discussion, the "MABEL" identifier ("l1" in the above example). This special case will be detected by the look-ahead algorithm. The MABEL identifier will be assumed to be a label-identifier unless it identifies the defining occurrence of a mode-identifier, in which case it will be assumed to be an applied occurrence of this mode-identifier and an entry for the MABEL identifier will be made in a special table. At any applied occurrence of a label-identifier, this special table will be inspected. If the table contains an entry that the applied occurrence of the label-identifier could identify as a defining occurrence, then the corresponding MABEL identifier will be assumed to be a label-identifier. The necessary modifications to tables will be made and back-tracking will occur in that the code generated under the assumption that the MABEL identifier was a mode-identifier, and the construct was part of an actual-declarer, must be replaced by the code that precedes entry to a range. Finally when the corresponding right parenthesis is processed, the immediate right context will verify or contradict the assumption and the entry for the MABEL identifier in the special table may be deleted. In the case of contradiction, an error message will ensue.

The major disadvantage of using specific criteria, such as those mentioned above, during compilation is that errors occurring in a program could be very misleading [40]. Difficulties in error handling are inherent in the language since there exist many syntactically similar but semantically different constructions. The error handling strategy to be used in this implementation is as follows: following the detection of an error, the source program will be scanned to the right until a symbol is found, which together with the left context already analyzed, delimits the section of program containing the error. The symbols which delimit error sections vary depending on the type of error. The matching of symbols, as required in the syntax of Appendix A, will be used in the search for error-delimiting symbols. No attempt will be made to correct errors occurring in a source program and no object code will be generated following the discovery of an error, though analysis will continue. Several techniques for reducing the number of consequential errors (that is, errors resulting from a previous error) are being investigated. Some errors, for example, doubly-defined identifiers, do not cause problems as far as delimiting is concerned but may cause many consequential errors. It would be an advantage to report at least the possibility that an error is consequential.

4.4 Table Structure

The following are the main tables of the proposed one-pass compiler.

1. Lexical analysis table - used by the lexical analyzer to generate unique integers for such tokens as "confrontation-tokens", "declaration-tokens", "syntactic-tokens", "sequencing-tokens", and "hip-tokens".
2. Indication and identifier table - contains actual sequences of characters used for identifiers and indications.
3. Standard-prelude table - contains entries for operators, indications, procedures, and some of the identifiers which are declared in the standard-prelude.
4. Main table - contains entries for each operator, indication, and identifier occurring within the program being compiled.
5. Declarer table - contains entries for each declarer occurring in the program being compiled.

Other tables will certainly exist within the compiler; for example, a table to assist in code generation will be necessary, but tables such as this will not be detailed in this investigation. The lexical analysis and standard-prelude tables are of fixed sizes. The other three tables mentioned above and the compiler-stack (used by the recursive descent routines) will vary in size depending on

the particular-program being compiled. If an overflow occurs, free storage will be redistributed so that the overflow condition disappears, if possible, according to an algorithm given in [41].

The main table will initially be empty. Any declaration of an operator, indication, or identifier will cause an entry to be made in this table. In order to reduce search time, entries will be made using hash addressing techniques with quadratic rehash to resolve collisions [23-Chapter 9]. The information contained in an entry in the main table depends on the type of entry. [39-Chapter 5] gives a detailed description of the type of information that will be stored in the main table. A scheme similar to that described in [27] is used to represent the static block structure of a program in order that correct identification of identifiers, indications, and operators may be made. A modification is necessary however, in that ranges are not immediately recognizable to a one-pass compiler during a left-to-right scan even with the look-ahead algorithm as given in the previous section. In [42] "corrals" are defined as constructs which have the same context as serial-clauses (including serial-clauses themselves) and it is proved that no errors will occur in the identification of indications, operators, or identifiers if all corrals except "formal-PARAMETERS" [R5.4.1.c] are considered as ranges. It is

possible to distinguish formal-PARAMETERS from other corral-
by use of the look-ahead algorithm. So, for the purposes of
identification of operators, indications, and identifiers,
the algorithm in [27] is used but with corral-
excluding formal-PARAMETERS, replacing ranges. In the sequel, when
"corral" is mentioned, it is to be assumed that formal-
PARAMETERS have been excluded.

When an applied occurrence of an operator, indication,
or identifier is encountered, the main table will be
searched for the corresponding defining occurrence. In the
case that a defining occurrence is not found then the
standard-prelude table will be searched (also using hash-
addressing). If the defining occurrence is found in this
table then the entry will be transferred to the main table.
In this way, only those operators, identifiers, and
indications declared in the standard-prelude which are used
in the particular-program being compiled will appear in the
main table. If the defining occurrence is not found in the
standard-prelude table then for indications and operators an
error message will be given; for identifiers a final
possibility exists. A routine will be called to check
whether the identifier is one of the infinity of identifiers
declared in the standard-prelude which, for practical
reasons, is not yet stored in the standard-prelude table;
for example, "long long long max int". If this check fails,
then an error message will be given.

As noted in Chapter III it is also necessary to record applied occurrences of indications, identifiers and operators so as to be able to report errors such as the one in the following example:

```
begin mode hippo = int;
      begin hippo x;
            mode hippo = real; .
```

At compile-time, whenever a new corral is encountered within the source program, the compiler will reserve a flag for each indication and identifier having a defining occurrence which could be identified by an applied occurrence in the corral. These flags will be stored in the main table with their corresponding indications and identifiers and will initially be set off (a flag may be "on" or "off"). When an applied occurrence of an indication or identifier is found to identify a defining occurrence, then the flag corresponding to that defining occurrence will be set on. When the compiler exits from a corral the set of flags for the corral being left will be "or"ed (in the Boolean sense) with the set of flags for the corral being re-entered. In this way a record will be kept of those indications and identifiers whose defining occurrences were identified by applied occurrences at deeper levels of nesting. Upon encountering the declaration of an indication or identifier which is the same sequence of marks as an already existing

and accessible defining occurrence (declared within an outer range), the flag for this existing defining occurrence in the current set of flags will be inspected. If it is on, then any applied occurrence already processed would (and should) have identified this new defining occurrence and an error message will be given. If the inspected flag is off, then the processing of the declaration will be completed.

The above algorithm cannot be applied to operators because of the complex mechanism of operator identification and the acceptance of the double-identification context condition, with the consequent abolition of the loosely related condition. The following example should illustrate this:

```

op rhino = (ref int i) void: skip;
begin int i;
    rhino i;
    op rhino = (real i) void: skip;
    op rhino = (int i) void: skip; .

```

In a left-to-right scan of the above example, the formula rhino i is found to contain an operator rhino which identifies the rhino declared in the outer range, and has an operand of reference-to-integral mode. Following this formula are two operation-declarations. The first of these declares an operator whose representation consists of the same sequence of marks as an already applicable defining occurrence but its operand is of real mode and hence this

declaration should not result in any error messages. The second declaration defines an operator whose representation also consists of the same sequence of marks. However its operand is of integral mode and now trouble arises. The operator of the formula rhino i should identify this defining occurrence since "i" can be dereferenced from reference-to-integral mode to integral mode. With the removal of the loosely related condition, it would now be impossible to find corresponding flags within the set of flags for operators, if these had been created. The solution involves the recording of all operator-applied occurrences and the modes of the operands before coercion. In the case that an operand is a "balance" [R6.2.1.e] or a collateral-clause, the modes of the components must be recorded. At the declaration of an operator, these applied occurrences must be inspected so as to determine whether the operator-defining occurrence in the new declaration could have been identified. This solution is very expensive and in the proposed implementation it will not be used. Rather, it is proposed to use a similar algorithm to that described above for indications and identifiers. A flag will be created for each operator with a distinct representation, and for each operator whose representation is the same as that of some other operators, but whose operand(s) are of different mode(s) than those of the other operators. Thus an error message will be given in the case of the declaration of an

operator which is the same symbol as, and whose operand(s) are of the same mode(s) as the operand(s) of, an already existing applicable operator-defining occurrence which has been identified by an operator-applied occurrence at a deeper level of nesting. A warning message will be given in the case of the declaration of an operator which is the same symbol as an already existing applicable operator-defining occurrence which has been identified by an operator-applied occurrence at a deeper level of nesting. In the above example, the two inner operation-declarations would thus produce warning messages. This method is chosen because it is much less expensive than the complete solution even though, technically speaking, the proposed compiler is not now a compiler for a sublanguage of ALGOL 68. It will be possible to write programs in the sublanguage, the elaboration of which will be different than that specified by the Report. For the chosen algorithm, the maximum number of flags necessary is equal to the maximum number of indications, identifiers, and operators times the maximum depth of nesting of corrals.

The task of representing modes in a suitable manner at compile-time is not trivial, because the set of modes is infinite and modes may be defined recursively. The declarer table is constructed in such a way as to enable the storing and efficient manipulation of modes [29, 39-Chapter 5, 43,

44, 45]. When processing the declarer of a mode-declaration the indication of which ultimately identifies (see Chapter III) an indication-applied occurrence whose indication-defining occurrence has not yet been encountered, then this declarer must be "remembered" so that when the missing indication-defining occurrence is encountered, processing of the declarer may be continued.

The algorithm for determining the "equivalence" of modes given in [44] is not immediately applicable in a one-pass compiler since this algorithm efficiently tests a set of modes already given, whereas in a one-pass compiler, declarers must be processed as they are encountered, rather than all at once. One alternative is to use an algorithm similar to those given in [29, 46] for the pair-wise testing of equivalence. However, another alternative to be considered arises from [47], where a method of ordering modes is outlined. An algorithm based on this method would order modes as the corresponding declarers were entered in the declarer table and equivalence could be checked at the same time. This ordering of modes has other advantages that will prove beneficial during compilation [47].

4.5 Coercions and Identification of Operators

The problem of determining coercion steps is not unique to ALGOL 68 but it is much more complex than in other

high-level languages because there are more coercions and because there is a potential infinity of modes in ALGOL 68.

The task of operator identification is also included in this section because of the role of coercions in this process. Initially the identification of operators proceeds analogously to the identification of indications. However, in ALGOL 68, the mode(s) of the operand(s) of an operator are also considered in the identification process. The mode(s) of the operand(s) of an applied occurrence of an operator ("a priori" modes) must be "firmly coercible" [R4.4.3.a] to the mode(s) of the formal-parameter(s) given in the corresponding operation-declaration ("a posteriori" modes). Operator identification is also complicated by "balancing" and by collateral-clauses. In the general problem there is a set (perhaps infinite) of a posteriori modes and a set of a priori modes. A unique sequence of coercions must be found from a mode in the a priori set to a mode in the a posteriori set such that constraints imposed by context are satisfied.

The processes of operator identification and determination of coercions form a major part of an ALGOL 68 compiler and the problems involved for a one-pass compiler seem to be no more complex than those involved for a multi-pass compiler. An investigation of implementation techniques for these processes within the proposed one-pass compiler

has not yet been completed. Discussion of the processes and algorithms for some of the tasks involved may be found in [12, 39-Chapter 7, 44, 48, 49, 50].

4.6 Code Generation

More or less standard techniques exist for synthesizing object code once the requisite information has been gathered during the analysis phase of compilation. Several problems remain for the one-pass compiler and these will be discussed in this section.

The proposed one-pass compiler will generate object code directly, locating it within the runtime environment so that when compilation has finished, execution may begin immediately. [39-Chapter 11] contains a detailed description of code generation for an ALGOL 68 compiler. Because of the availability of compile-time tables at runtime, it is possible that some of the runtime routines might be interpretive; for example, routines for the elaboration of declarers. Transput routines will be based on algorithms presented in [51] and thus format-denotations will be compiled as suggested in that reference. Only very local optimizations will be made during code generation.

Several runtime options will be available; for example, initialization-before-use-checking, bound-checking, state-checking, and scope-checking as well as diagnostic and

debugging aids. These checks are provided as options because a correct proper particular-program in the sublanguage will not require them. Of these checks, scope-checking warrants the most attention and a discussion of scope-checking for a multi-pass compiler may be found in [39-Chapter 8], the problem not having been investigated as yet for this one-pass implementation.

One of the problems that a one-pass compiler must solve is that of code generation for jumps. Often, when an applied occurrence of a label-identifier is processed, the defining occurrence which it identifies will not yet have been encountered, or it may have been encountered but in an outer range, in which case the compiler cannot be certain whether or not it really is the defining occurrence. Therefore, special action using indirect branching is sometimes necessary. If the defining occurrence of the constituent label-identifier of a jump has already been encountered within the same range as the jump, then code for the jump may be generated directly. Otherwise, a branch instruction which transfers control to some reserved location in the "coffin" (described in section seven of this Chapter), will be generated, and an entry for the applied occurrence of the label-identifier within the jump will be created in the main table. This entry will be flagged as not having been completed and will contain a pointer to the

reserved location in the coffin and a pointer to a defining occurrence (in an outer range) it might identify, if one exists. When the corresponding defining occurrence is encountered or verified (by re-entering the range containing the defining occurrence that the applied occurrence might identify) then the transfer point is known and a branch instruction to this transfer point will be inserted in the reserved location, thus completing the indirect branch. Many slight modifications of this approach exist, depending on the instruction set of the physical machine for which the compiler is being implemented.

Another problem peculiar to a one-pass compiler is the generation of code for coercions. In general, when a coerced is encountered it is not known what coercions, if any, are to be performed until further source text is processed. The preceding coercion affects whether or not a coerced is elaborated and thus provision must be made for the situation when a given coerced is not elaborated, but proceeded. All other coercions follow elaboration of the coerced, and provision must be made for inserting code for these or at least branching to the code. Instead of detailing the method of solution, three examples will be given, illustrating the problems and the techniques used to solve them.

Example 1:

This example illustrates why coercions are not, in general,

found until it is too late to insert the code for them directly in place in the object program. Consider a opr b, where "a" and "b" represent any coerccends, and opr represents any operator. First the coerccend "a" will be processed and an a priori mode (or modes) obtained. The operator opr will then be scanned but, in general, it will not yet be known what operator-defining occurrence this operator-applied occurrence identifies. Thus the coerccend "b" must be processed in order to obtain the a priori mode (or modes) of the right operand so that operator identification may be completed. Once the identification is complete then and only then will the coerccions for "a" and "b" be known.

Example 2:

Consider a coerccend in a context such that dereferencing and then widening must occur. Let "a" denote the coerccend and "code(a)" the code generated for the elaboration of "a". Then the following sequence of "pseudo-instructions" represents the flow of the code that will be generated.

- 1: Jump to 2.
- 2: Code(a).
- 3: Jump to 6.
- 4: Perhaps other code (generated while the compiler discovers the coerccions required for "a").
- 5: Jump to 9.
- 6: Dereference(2) ("2" represents the result of

elaborating code(a)).

7: Widen(6) .

8: Jump to 4.

9: ...

The transfer points for jumps and the locations from which jumps are made may be remembered by employing stack techniques.

Example 3:

Consider a coerced in a context such that widening, proceduring, and rowing must occur. "a" and "code(a)" are as in example 2. The following sequence of pseudo-instructions represents the flow of the code that will be generated.

1: Jump to 10.

2: Code(a) .

3: Jump to 6.

4: Perhaps other code.

5: Jump to 13.

6: Widen(2) .

7: Epilog.

8: Prolog.

9: Jump to 2.

10: Routine builder(8) .

11: Row(10) .

12: Jump to 4.

13: ...

"Prolog" and "epilog" represent standard code generated for

procedure entry and exit. "Routine builder" takes the location of the procedured coerced ("8" in the above) and builds a representation for the procedure (environment information must be added).

4.7 Runtime Environment

Little detail will be given in this discussion of the runtime environment, again because investigation is incomplete and because there exist methods proposed for multi-pass compilers that will, with little modification, suffice for the proposed one-pass compiler.

When an ALGOL 68 object program is running, three separate parts of storage may be distinguished.

1. The "coffin" contains object-code and constant data for the program.
2. The "stack", on which is allocated the space for most locally declared objects, for intermediate results, and for organizational entities.
3. The "heap", on which space is allocated for all global generations, and for local objects whose storage requirements may vary during elaboration.

The stack and the heap will occupy opposite ends of an area of storage and grow towards each other as space is allocated on each of them. The recovery of space from the

stack will be achieved using stack techniques similar to those of ALGOL 60 implementation [23-Chapter 8, 25]. Space will be recovered from the heap whenever a collision between the stack and heap is imminent. The process of recovering space from the heap is known as garbage-collection and techniques for this are described in detail in [27, 39-Chapter 10, 52].

The stack will be organized procedure-wise, that is, the stack will consist of blocks of storage called stack cells, each of which is the storage local to a particular invocation of a routine. [39-Chapter 10, 53] contain further details of runtime environment including the representation of values and detailed stack organization.

The runtime environment will include facilities for runtime debugging and post-mortem dumps. If requested, a modified copy of the source program will be retained during elaboration of the object program. This copy of the source program may be compacted somewhat because the compiler tables will also be retained, remembering that the implementation is a compile-and-go system. Traces, statement counts, and post-mortem dumps will then be made available to the user in a similar fashion to that described in [54].

CHAPTER V

CONCLUSION

The objectives of this investigation were twofold. Firstly, a syntax for a modified version of ALGOL 68 was developed. This syntax, together with a corresponding context-free syntax and a representation language, appear as Appendices to this thesis. The modifications made to the language described in the Report, consist of additions and alterations considered to enhance the language from a user's, as well as an implementor's point of view. Included among the modifications were some of the many suggestions for additions and alterations proposed since the publishing of the Report, as well as some changes arising out of this investigation. The modifications and reasons for their incorporation are discussed in Chapter II of this thesis. Some suggestions for semantic changes are included but emphasis is generally on syntactic rather than semantic details.

The second objective of this thesis was the design of a one-pass compiler for ALGOL 68. Implementation

dependencies and restrictions imposed to facilitate the one-pass implementation are discussed in Chapter III. Chapter IV describes techniques for some of the tasks which the compiler must perform. Experience gained while designing the compiler was of considerable importance in the consideration of modifications to the language.

ALGOL 68 is a powerful programming language incorporating and generalizing many concepts from other programming languages and from current literature. As well as this, many new features are introduced and these give rise to difficulties hitherto unencountered in the implementation of programming languages. Implementation of ALGOL 68 is thus more difficult and more expensive than, for example, the implementation of ALGOL 60. Chapter IV of this thesis is by no means complete in its description of the proposed one-pass compiler and there remain many problems on which further research must be carried out. These problems however, are not just peculiar to one-pass compilers but to multi-pass compilers as well. At this stage it would seem that a one-pass compiler for the given sublanguage of ALGOL 68 may be constructed, provided that a multi-pass compiler for the unrestricted language is possible. The quality of object code produced by this one-pass compiler will be poor when compared with that code which could be produced by a multi-pass compiler. This is because of the extra

information available to the multi-pass compiler when generating object code.

It is proposed to implement a one-pass compiler based on the design presented in this thesis. The compiler will be written in a low-level language of a particular physical machine. A low-level language is chosen so that efficiency of the compiler is improved, despite the increases in programming and debugging time. There are two proposed usages for the compiler. Firstly, it will be used in an educational environment providing fast turn-around for normal, small-to-medium size programs, together with good debugging and diagnostic facilities. Secondly, the compiler will be used to experiment with the inclusion and exclusion of various features of ALGOL 68, thus giving some idea of the cost, with respect to both efficiency and size, of including certain of these features in ALGOL 68 and its sublanguages.

REFERENCES

1. Van Wijngaarden, A. (editor), Mailloux, B.J., Peck, J.E.I., and Koster, C.H.A., Report on the Algorithmic Language ALGOL 68, Report MR101, Mathematisch Centrum, Amsterdam, 1969.
2. Naur, P. (editor), "Revised Report on the Algorithmic Language ALGOL 60", Communications of the ACM, VI, (January, 1963), p. 1.
3. WG2.1 Formal Resolution: "Revised Report on ALGOL 68", ALGOL Bulletin, No. 33, (March, 1972), p. 3.
4. Lindsey, C.H., Proposals to the Brussels Meeting of the Subcommittee on Improvements of the Working Group 2.1 of IFIP.
5. Lindsey, C.H., Proposals to the Malvern Meeting of the Subcommittee on Improvements of the Working Group 2.1 of IFIP.
6. Report on Considered Improvements, IFIP Working Group 2.1 Paper (Fontainebleau 9) 192.
7. Lindsey, C.H., and van der Meulen, S.G., Informal Introduction to ALGOL 68, (North Holland, 1971).
8. Bocm, H.J., Some Proposals, IFIP Working Group 2.1 Paper (Fontainebleau 4) 197, (February, 1972).
9. Broderick, W.R., Letter to the Editor, ALGOL Bulletin, No. 28, (July, 1968), p. 4.
10. Freeman, W., Suggestions Regarding Certain Representations in ALGOL 68, IFIP Working Group 2.1 Paper (Fontainebleau 16) 199.
11. Report from University of Alberta ALGOL 68 Implementors, IFIP Working Group 2.1 Paper (Novosibirsk 11) 176.

12. Woessner, H., "On Identification of Operators in ALGOL 68", in ALGOL 68 Implementation, J.E.L. Peck (editor), (North-Holland, 1971), p. 111.
13. WG2.1: "Report of the Subcommittee on Maintenance of and Improvements to ALGOL 68", ALGOL Bulletin, No. 32, (May, 1971), p. 40.
14. Thomas, I.K., and Wilmott, S.J., Comments on Current Proposals being Considered by the Subcommittee on Maintenance and Improvement (Revised), IFIP Working Group 2.1 Paper (Fontainebleau 3) 186, (June, 1972).
15. Report of the Subcommittee on Improvements, IFIP Working Group 2.1 Paper (Habay 15) 157.
16. Hill, U., Scheidig, H., and Woessner, H., ALGOL 68 M, Technical University of Munich, Report Nr. 7009.
17. van der Poel, W.L., The Engelfriet-Rijpens Ambiguity, IFIP Working Group 2.1 Paper (Fontainebleau 14) 197.
18. Bourne, S.R., and Guy, M.J.T., "Suggestions for Improvements to ALGOL 68", ALGOL Bulletin, No. 33, (March, 1972), p. 47.
19. Golde, H., Extension 9.2a and the Syntax of Generators, IFIP Working Group 2.1 Paper (Novosibirsk 8) 173.
20. Hill, U., and Woessner, H., Comments, IFIP Working Group 2.1 Paper (Fontainebleau 2) 185.
21. Hodgson, G.S., ALGOL 68 Extended Syntax, Department of Computer Science, The University, Manchester, (March, 1970).
22. Hopgood, F.R.A., Compiling Techniques, (MacDonald/American Elsevier, 1969).
23. Gries, David, Compiler Construction for Digital Computers, (Wiley, 1971).
24. Glass, R.I., "An Elementary Discussion of Compiler/Interpreter Writing", Computing Surveys, I, (March, 1969), p. 55.

25. Randell, B., and Russell, L.J., ALGOL 60 Implementation, (Academic Press, 1964).
26. WG2.1 Subcommittee: "Report on Sublanguages", ALGOL Bulletin, No. 33, (March, 1972), p. 43.
27. Mailloux, B.J., "On the Implementation of ALGOL 68", (Ph.D. Dissertation, Mathematisch Centrum, Amsterdam, 1968).
28. Lindsey, C.H., "Some ALGOL 68 Sublanguages", in ALGOL 68 Implementation, J.E.L. Peck (editor), (North-Holland, 1971), p. 283.
29. Peck, J.E.L., "On Storage of Modes and Some Context Conditions", in Proceedings of an Informal Conference on ALGOL 68 Implementation, (Department of Computer Science, University of British Columbia, 1969), p. 70.
30. Weissman, C., LISP 1.5 Primer, (Dickenson, 1967).
31. Griswold, R.E., Poage, J.F., and Polonsky, I.P., The SNOBOL 4 Programming Language, (Prentice-Hall, 1968).
32. Johnson, W.L., et al., "Automatic Generation of Efficient Lexical Processors Using Finite State Techniques", Communications of the ACM, XI, (December, 1968), p. 805.
33. Floyd, R.W., "Bounded Context Syntactic Analysis", Communications of the ACM, VII, (February, 1964), p. 62.
34. Koster, C.H.A., "Affix-Grammars", in ALGOL 68 Implementation, J.E.L. Peck (editor), (North-Holland, 1971), p. 95.
35. Branquart, P., Lewi, J., and Cardinael, J.P., A Context-Free Syntax of ALGOL 68 (Revised version), Technical Note N66, M.B.L.E. Research Lab., Brussels, (August, 1970).
36. Floyd, R.W., "A Descriptive Language for Symbol Manipulation", Journal of the ACM, VIII, (October, 1961), p. 579.

37. Evans, A., "An ALGOL 60 Compiler", Annual Review in Automatic Programming, 4, (1964), p. 87.
38. Westland, J., "An ALGOL 68 Syntax and Parser", (M.Sc. Thesis, University of Calgary, 1969).
39. Hill, U., Scheidig, H., and Woessner, H., An ALGOL 68 Compiler, Technical Report, Technical University of Munich, University of British Columbia.
40. Branquart, P., and Lewi, J., Analysis of the Parenthesis Structure of ALGOL 68, Report R130, M.B.L.E. Research Lab., Brussels, (April, 1970).
41. Knuth, D.E., Fundamental Algorithms, Vol. I of The Art of Computer Programming, (Addison-Wesley, 1968).
42. Koch, F.H., "The Recognition of Ranges in ALGOL 68", (M.Sc. Thesis, University of Calgary, 1969).
43. Andrews, M.P., "Practical Considerations in the Storage of Modes", in Proceedings of an Informal Conference on ALGOL 68 Implementation, (Department of Computer Science, University of British Columbia, 1969), p. 78.
44. Zosel, M.E., "A Formal Grammar for the Representation of Modes and its Application to ALGOL 68", (Ph.D. Dissertation, University of Washington, 1971).
45. Peck, J.E.L., Manipulation of Modes in ALGOL 68, IFIP Working Group 2.1 Paper (Fontainebleau 11) 194.
46. Koster, C.H.A., "On Infinite Modes", ALGOL Bulletin, No. 30, (February, 1969), p. 86.
47. Wilmott, S.J., "On the Ordering of Modes in ALGOL 68", (Forthcoming M.Sc Thesis, University of Alberta).
48. Branquart, P., and Lewi, J., On the Implementation of Coercions in ALGOL 68, Report R123, M.B.L.E. Research Lab., Brussels, (January, 1970).
49. Scheidig, H., Coercions in ALGOL 68, Technical University of Munich.
50. Peck, J.E.L., and Kwan, Ying, Operator Identification Procedures, Informal ALGOL 68 Meeting, University of British Columbia, (June, 1972).

51. Lyall, C.J.C., "Implementation of Transput for an ALGOL 68 Compiler", (Forthcoming M.Sc. Thesis, University of Alberta).
52. Branquart, P., and Lewi, J., "A Scheme of Storage Allocation and Garbage Collection for ALGOL 68", in ALGOL 68 Implementation, J.E.L. Peck (editor), (North-Holland, 1971), p. 199.
53. Wilmott, S.J., General Storage Organization of an ALGOL 68 Program, IFIP Working Group 2.1 Paper (Malvern 12) 152.
54. Sites, R.I., "Deck Setup and Compiler Options", in ALGOL W Reference Manual, Report STAN-CS-71-230, Stanford University, (August, 1971), p. 102.

APPENDIX A

A MODIFIED ALGOL 68 SYNTAX

This syntax is presented in the same order as that of the Report, with corresponding section numbers and headings. Vertical lines in the left margin denote those parts of the syntax that are additions or alterations. Omissions in the ordering sequence correspond to productions present in the Report but not included in this syntax.

1.2.1. Metaproduction Rules of Modes

- a) MODE: MCOD; UNITED.
- b) MOOD: TYPE; STOWED.
- c) TYPE: PLAIN; format; PROCEDURE; reference to MCDE.
- d) PLAIN: INTREAL; boolean; character.
- e) INTREAL: INTEGRAL; REAL.
- | f) INTEGRAL: SHCNGSETY integral.
- | g) REAL: SHCNGSETY real.
- | ha) SHONGSETY: long LONGSETY; short SHORTSETY; EMPTY.
- | hb) LONGSETY: long LONGSETY; EMPTY.
- | hc) SHORTSETY: short SHORTSETY; EMPTY.
- | hd) SHCNG: short; long.
 - i) EMPTY: .
 - j) PROCEDURE: procedure PARAMETY MOID.
 - k) PARAMETY: with PARAMETERS; EMPTY.
 - l) PARAMETERS: PARAMETER; PARAMETERS and PARAMETER.
- ma) PARAMETER: MCDE parameter.
- | mb) PARAMETERS: parameter and MODE PARAMETERS; parameter.
 - n) MOID: MCDE; void.
- | o) STCWED: structured with FIELDS; ROWS of MODE.
 - p) FIELDS: FIELD; FIELDS and FIELD.
 - qa) FIELD: MODE field TAG.

- |qb) FOLDS: field TAG and FIELDS; field TAG.
- r) TAG: LETTER; TAG LETTER; TAG DIGIT.
- s) LETTER: letter ALPHA; letter aleph.
- t) ALPHA: a; b; c; d; e; f; g; h; i; j; k; l; m; n; o; p;
q; r; s; t; u; v; w; x; y; z.
- u) DIGIT: digit FIGURE.
- v) FIGURE: zero; one; two; three; four; five; six; seven;
eight; nine.
- w) UNITED: union of LMOODS MOOD mode.
- x) LMOODS: LMCOD; IMOCDS IMOOT.
- y) LMCCD: MOOD and.

1.2.2. Metaproduction Rules Associated with Modes

- |ba) ROWS: row; RCWS row.
- |bb) ROW: row; row of.
- c) ROWSETY: RCWS; EMPTY.
- e) NONROW: NCNSTOWED; structured with FIELDS.
- f) NCNSTOWED: TYPE; UNITED.
- g) REFETY: reference to; EMPTY.
- | h) NCNPROC: PLAIN; format; reference to NCNPRCC;
procedure with PARAMETERS MOID; UNITED;
structured with FIELDS; row of MODE.
- i) PRAM: procedure with RMODE parameter MOID;
procedure with LMODE parameter and RMCDE parameter
MOID.
- j) LMCDE: MODE.
- k) RMCDE: MODE.
- l) LMCCD: MCCD and.
- m) LMOODSETY: MOOD and LMCODSETY; EMPTY.
- n) RMOCDSETY: RMOODSETY and MOOD; EMPTY.
- o) LOSETY: LMOODSETY.
- p) BOX: IMCCDSETY box.
- q) LFIELDSETY: FIELDS and; EMPTY.
- r) RFIELDSETY: and FIELDS; EMPTY.
- s) CCMFLEX: structured with real field letter r letter e
and real field letter i letter m.
- | t) BITS: structured with row of boolean field SHCNGTHETY
letter aleph.
- |ua) SHCNGTHETY: LENGTH LENGTHETY; SHORTH SHCRTHETY; EMPTY.
- |ub) LENGTHETY: LENGTH LENGTHETY; EMPTY.
- |uc) SHORTHETY: SHORTH SHORTHETY; EMPTY.
- va) LENGTH: letter l letter o letter n letter g.
- |vb) SHORTH: letter s letter h letter o letter r letter t.
- | w) BYTES: structured with row of character field
SHCNGTHETY letter aleph.
- x) STRING: row of character; character.
- y) MABFI: MODE mode; label.

1.2.3. Metaproduction Rules Associated with Phrases and Coercion

- a) PHRASE: declaration; C1AUSE.
- b) C1AUSE: MCID clause.
- | c) SCME: serial; unitary; CLOSED; choice; chooser; in; out; repetitive; FROBYT; while; do.
- | d) CLCSED: closed; collateral; CONFASE.
- | f) CCNFASE: conditional; case; conformity case.
- g) SORT: strong; FEAT.
- | h) FEAT: firm; meek; weak; soft.
- j) STIRM: strong; firm.
- k) ADAPTED: ADJUSTED; widened; rowed; hippled; voided.
- l) ADJUSTED: FITTED; procedured; united.
- m) FITTED: dereferenced; deprocedured.
- | n) CCNFETY: conformity; EMPTY.
- | o) FRCBYT: from; by; to.

1.2.4. Metaproduction Rules Associated with Coercends

- a) COERCEND: MCID FORM.
- b) FORM: confrontation; FCRESE.
- | c) FORESE: PRICRETY ADIC formula; cohesion; base; routine denotation.
- | d) ADIC: dyadic; monadic.
- ea) PRICRITY: priority NUMBER.
- | eb) PRIORITIES: PRIORITY; priority NINE plus one.
- | ec) PRICRETY: PRIORITIES; EMPTY.
- f) NUMBER: one; TWO; THREE; FOUR; FIVE; SIX; SEVEN; EIGHT; NINE.
- g) TWC: one plus one.
- h) THREE: TWO plus one.
- i) FOUR: THREE plus one.
- j) FIVE: FCUR plus one.
- k) SIX: FIVE plus one.
- l) SEVEN: SIX plus one.
- m) EIGHT: SEVEN plus one.
- n) NINE: EIGHT plus one.

1.2.5. Other Metaproduction Rules

- a) VICTAI: VIRACT; formal.
- ka) VIRACT: virtual; actual.
- | bb) VIRMAI: virtual; formal.
- c) LOWPER: lower; upper.
- d) ANY: KIND; suppressible KIND; replicatable KIND; replicatable suppressible KIND.
- e) KIND: sign; zero; digit; point; exponent; complex; string; character.
- f) NOTICN: ALPHA; NOTICN ALPHA.
- | g) SEPARATOR: LIST separator; go on tokyl; completer;

sequencer.

- h) LIST: list; sequence.
- i) PACK: pack; package.
- | j) BRACKET: bracket; packet; pack.
- | k) RALIX: two; four; eight; sixteen.
- | l) MATCH: stop; brief.

2.1. The Computer and the Program

- a) program: open symbol, standard prelude, library prelude option, particular program, exit, library postlude option, standard postlude, close symbol.
- b) standard prelude: declaraticn prelude sequence.
- c) library prelude: declaration prelude sequence.
- | d) particular program: strong serial void clause PACK, comment sequence option.
- e) exit: go on symbol, letter e letter x letter i letter t, label symbol.
- f) library postlude: statement interlude.
- g) standard postlude: strong void clause train.

3.0.1. General Constructions

- b) NOTION opticon: NOTION; EMPTY.
- c) chain of NOTIONS separated by SEPARATORS: NOTION; NOTICN, SEPARATOR, chain of NOTIONS separated by SEPARATORS.
- d) NOTION LIST: chain of NOTIONS separated by LIST separators.
- | e) list separator: comma tokyl.
- f) sequence separator: EMPTY.
- g) NOTION LIST proper: NOTICN, LIST separator, NOTION LIST.
- | h) NOTION pack: open tokyl, NOTION, close tokyl.
- | i) NOTION package: begin tokyl, NOTION, end tokyl.
- | j) NOTION bracket: sub tokyl, NOTION, bus tokyl.
- | k) NOTION packet: alternate sub tokyl, NOTION, alternate bus tokyl.
- | l) NOTION tokyl: comment sequence option, NOTION symbol.

3.0.2. Letter Tokens

- | b) LETTER: LETTER tokyl.

3.0.3. Digit Tokens

- c) digit token: DIGIT.
- | d) DIGIT: DIGIT tokyl.

3.0.9. Comments

- | b) comment: MATCH comment tokyl, comment item sequence option, MATCH comment symbol.
- | c) comment item: character token; other comment item.
- | d) character token: LETTER symbol; DIGIT symbol; point symbol; times ten to the power symbol; open symbol; close symbol; comma symbol; space symbol; plus symbol; minus symbol.

4.1.1. Identifiers

- a)* identifier: MABEL identifier.
- b) MABEL identifier: TAG.
- c) TAG LETTER: TAG, LETTER.
- d) TAG DIGIT: TAG, DIGIT.
- | e)* range: program; SORT serial CLAUSE; procedure PARAMETY MOID routine denotation; SORT MOID conformity unit; do loop part.

4.2.1. Indications

- | a)* indication: MODE mode indication; PRIORETY ADIC indication.
- | b) MODE mode indication: mode standard; comment sequence option, indicant.
- | c) mode standard: boolean tokyl; character tokyl; format tokyl; string tokyl; sema tokyl; file tokyl; SHONG tokyl sequence option, integral tokyl; SHONG tokyl sequence option, real tokyl; SHONG tokyl sequence option, complex tokyl; SHONG tokyl sequence option, bits tokyl; SHONG tokyl sequence option, bytes tokyl.
- | d) PRIORITIES ADIC indication: ADIC indication.
- | e) ADIC indication: SHONG tokyl sequence option, comment sequence option, ADIC indicant.
- | g)* adic indication: PRIORETY ADIC indication.

4.3.1. Operators

- | a)* operator: PRAM PRIORETY ADIC operator.
- | b) PRAM PRIORETY ADIC operator: PRIORETY ADIC indication.
- | d)* ADIC operator: PRAM PRIORETY ADIC operator.

5.0.1. Denotations

- a)* denotation: PLAIN denotation; BITS denotation; row of character denotation; format denotation; procedure PARAMETY MOID denotation.

5.1.0.1. Plain Denotations

- a)* plain denotation: PLAIN denotation.
- | b) SHONG INTREAL denotation:
SHONG tokyl, INTREAL denotation.

5.1.1.1. Integral Denotations

- a) integral denotation: digit token sequence.

5.1.2.1. Real Denotations

- a) real denotation:
variable point numeral; floating point numeral.
- b) variable point numeral:
integral part option, fractional part.
- c) integral part: integral denotation.
- | d) fractional part: point tokyl, integral denotation.
- e) floating point numeral: stagnant part, exponent part.
- f) stagnant part:
integral denotation; variable point numeral.
- g) exponent part:
times ten to the power choice, power of ten.
- | h) times ten to the power choice:
times ten to the power tokyl; letter e.
- i) power of ten: plusminus option, integral denotation.
- | j) plusminus: plus tokyl, minus tokyl.

5.1.3.1. Boolean Denotations

- | a) boolean denotation: true tokyl; false tokyl.

5.1.4.1. Character Denotations

- | a) character denotation:
MATCH quote tokyl, string item, MATCH quote symbol.
- b) string item:
character token; quote image; other string item.
- | c) quote image: MATCH quote symbol, MATCH quote symbol.

5.2.1. Bits Denotations

- a)* bits denotation: BITS denotation.
- | ba) structured with row of boolean field LENGTH LENGTHETY
letter aleph denotation:
long tokyl, structured with row of boolean field
LENGTHETY letter aleph denotation.
- | bb) structured with row of boolean field SHCRTH SHORTHETY
letter aleph denotation:
short tokyl, structured with row of boolean field
SHCRTHETY letter aleph denotation.

- | c) structured with row of boolean field letter aleph denotation:
RADIX radix, letter r, RADIX byte sequence.
- | d) two radix: digit two.
- | e) four radix: digit four.
- | f) eight radix: digit eight.
- | g) sixteen radix: digit one, digit six.
- | h)* byte: RADIX byte.
- | i) two byte: digit zero; digit one.
- | j) four byte: two byte; digit two; digit three.
- | k) eight byte: four byte; digit four; digit five; digit six; digit seven.
- | l) sixteen byte:
eight byte; digit eight; digit nine; letter a;
letter b; letter c; letter d; letter e; letter f.

5.3.1. String Denotations

- a)* string denotation: row of character denotation.
- | b) row of character denotation: MATCH quote tokyl, string item sequence proper option, MATCH quote symbol.

5.4.1. Routine Denotations

- | a)* routine denotation:
procedure PARAMETY MOID routine denotation.
- | ba) procedure MOID routine denotation: virtual MCID declarer, routine tckyl, strong MOID unit.
- | bb) procedure with PARAMETERS MOID routine denotation: formal PARAMETERS pack, virtual MOID declarer, routine tokyl, strong MOID unit.
- | c) VIRACT PARAMETERS and PARAMETER:
VIRACT PARAMETERS, comma tokyl, VIRACT PARAMETER.
- | ea) formal MODE PRAMETERS: formal MODE declarer, formal MODE PRAMETERS definition.
- | eb) formal MODE parameter and MODE PRAMETERS definition: MODE mode identifier, ccomma tckyl, formal MODE PRAMETERS definition.
- | ec) formal MODE parameter definition: MODE mode identifier.
- | ed) formal MODE parameter and RMODE PRAMETERS definition: MODE mode identifier, comma tokyl, formal RMODE PRAMETERS.
- f)* VICTAL parameters pack: VICTAL PARAMETERS pack.

5.5.1. Format Denotations

- | a) format denotation:
formatter tokyl, collection list, formatter tokyl.
- b) collection: picture; insertion option, replicator, collection list pack, insertion option.
- c) picture: MODE pattern option, insertion option.

- d) insertion: literal option, insert sequence; literal.
- e) insert: replicator, alignment, literal option.
- f) replicator: replication option.
- g) replication: dynamic replication; integral denotation.
- h) dynamic replication:
 - letter n, strong CLOSED integral clause.
- i) alignment:
 - letter k; letter x; letter y; letter l; letter p.
- j) literal: replicator, STRING denotation, replicated
 - literal sequence option.
- k) replicated literal: replication, STRING denotation.
- l) sign mould: loose replicatable zero frame, sign frame;
 - loose sign frame.
- m) loose ANY frame: insertion option, ANY frame.
- n) replicatable ANY frame: replicator, ANY frame.
- o) zero frame: letter z.
- p) sign frame: plusminus.
- q) suppressible ANY frame: letter s option, ANY frame.
- r) * frame: ANY frame.

5.5.2. Syntax of Integral Patterns

- | a) integral pattern: sign mould option, integral mould;
 - integral choice pattern.
- d) integral mould: loose replicatable suppressible digit
 - frame sequence.
- e) digit frame: zero frame; letter d.
- f) integral choice pattern:
 - insertion option, letter c, literal list pack.

5.5.3. Syntax of Real Patterns

- a) real pattern: sign mould option, real mould;
 - floating point mould.
- b) real mould: integral mould, loose suppressible point
 - frame, integral mould option;
 - loose suppressible point frame, integral mould.
- | c) point frame: point tokyl.
- d) floating point mould:
 - stagnant mould, loose suppressible exponent frame,
 - sign mould option, integral mould.
- e) stagnant mould: sign mould option, INTREAL mould.
- f) exponent frame: letter e.

5.5.4. Syntax of Boolean Patterns

- a) boolean pattern: insertion option, letter b, boolean
 - choice mould option.
- | b) boolean choice mould: open tokyl, literal, comma
 - tokyl, literal, close tokyl.

5.5.5. Syntax of Character Patterns

- a) character pattern: loose suppressible character frame.
- b) character frame: letter a.

5.5.6. Syntax of Complex Patterns

- a)* complex pattern: COMPLEX pattern.
- b) COMPLEX pattern: real pattern, loose suppressible complex frame, real pattern.
- c) complex frame: letter i.

5.5.7. Syntax of String Patterns

- a)* string pattern: row of character pattern.
- b) row of character pattern: loose string frame;
loose replicatable suppressible character frame
sequence proper;
insertion option, replication, suppressible
character frame.
- c) string frame: letter t.

5.5.7A. Syntax of Bits Patterns

- | a)* bits pattern: structured with row of boolean field
letter aleph pattern.
- | b) structured with row of boolean field letter aleph
pattern: radix mould, integral mould.
- | c) radix mould: insertion option, RADIX radix, letter r.

5.5.8. TransformatS

- a) structured with row of character field letter aleph
digit one transformat: firm format unit.

6.0.1. Phrases

- | a)* SCME phrase: SORT SOME PHRASE.
- | b)* SCME expression: SORT SOME MODE clause.
- | c)* SCME statement: strong SOME void clause.
- | d)* MODE constant: MODE FORM.
- | e)* MODE variable: reference to MODE FORM.
- | f)* procedure: REFETY PROCEDURE FORM.
- | g)* structure display: STIRM collateral structured with
FIELDS and FIELD clause.
- | h)* row display: STIRM collateral ROW MODE clause.

6.1.1. Serial Clauses

- | a) SORT serial CLAUSE: declaration prelude sequence option, suite of SORT CLAUSE trains.
- | b) declaration prelude: statement prelude option, single declaration, go on tokyl.
- | c) statement prelude: chain of strong void units separated by go cn tokyls, go on tokyl.
- d) single declaration:
 - unitary declaration; collateral declaration.
- | e) SORT MOID unit: SORT unitary MOID clause.
- | f) suite of strong CLAUSE trains: chain of strong CLAUSE trains separated by completers.
- g) suite of FEAT CLAUSE trains: FEAT CLAUSE train;
 - FEAT CLAUSE train, completer, suite of strong CLAUSE trains;
 - strong CLAUSE train, completer, suite of FEAT CLAUSE trains.
- | h) SORT MOID clause train: label sequence option, statement interlude option, SORT MOID unit.
- i) statement interlude: chain of strong void units separated by sequencers, sequencer.
- | j) sequencer: go on tokyl, label sequence option.
- | k) label: label identifier, label tokyl.
- | l) completer: completion tokyl, label.

6.2.1. Collateral Phrases

- a) collateral declaration:
 - unitary declaration list proper.
- b) strong collateral void clause: parallel symbol option, strong void unit list proper PACK.
- | c) STIRM collateral ROW MODE clause:
 - STIRM MODE balance PACK.
- | ea) strong MCID CONFETY balance:
 - strong MOID CONFETY unit list proper.
- | eb) FEAT MOID CCONFETY balance: FEAT MOID CCONFETY unit, comma tokyl, strong MOID CONFETY unit list;
 - strong MCID CONFETY unit, comma tokyl, FEAT MCID CONFETY unit;
 - strong MOID CONFETY unit, comma tokyl, FEAT MOID CCONFETY balance.
- | f) STIRM collateral structured with FIELDS and FIELD structure clause: STIRM structured with FIELDS and FIELD structure PACK.
- | g) STIRM structured with FIELDS and FIELD structure:
 - STIRM structured with FIELDS structure, comma tokyl, STIRM structured with FIELD structure.
- | h) STIRM structured with MODE field TAG structure:
 - STIRM MODE unit.

6.3.1. Closed Clauses

- | a) SORT closed CLAUSE: SORT serial CLAUSE PACK.

6.4.1. Conditional Clauses

- | a) SORT CCNFASE CLAUSE:
MATCH CCNFASE begin tokyl, SORT MATCH CCNFASE
chooser CLAUSE, MATCH CCNFASE end tokyl.
- | b) SORT MATCH CCNFASE chooser CLAUSE:
CCNFASE, SORT MATCH CCNFASE choice CLAUSE.
- | c) conditional: strong serial boolean clause.
- | d) case: strong serial integral clause.
- | e) conformity case: weak serial UNITED clause.
- | f) SORT MATCH conditional choice CLAUSE: MATCH thef
tokyl, SORT MATCH conditional chooser CLAUSE.
- | g) strong MATCH CCNFASE choice CLAUSE:
strong MATCH CCNFASE in CLAUSE, strong MATCH
CCNFASE out CLAUSE option.
- | h) FEAT MATCH CCNFASE choice CLAUSE: FEAT MATCH CCNFASE
in CLAUSE, strong MATCH CCNFASE out CLAUSE option;
strong MATCH CCNFASE in CLAUSE, FEAT MATCH CCNFASE
out CLAUSE.
- | i) SORT MATCH conditional in CLAUSE:
MATCH conditional in tokyl, SORT serial CLAUSE.
- | j) SORT MATCH case in MOID clause:
MATCH case in tokyl, SORT MOID balance.
- | k) SORT MATCH conformity case in MOID clause:
MATCH conformity case in tokyl, SORT MOID
conformity unit;
MATCH conformity case in tokyl, SORT MOID
conformity balance.
- | l) SORT MATCH CCNFASE out CLAUSE:
MATCH CCNFASE out tokyl, SORT serial CLAUSE;
MATCH CCNFASE out begin tokyl, SORT MATCH CCNFASE
chooser CLAUSE.
- | m) SORT MOID conformity unit:
MODE specification, SORT MOID unit.
- | n) MODE specification:
open tokyl, formal MODE declarer, MODE mode
identifier option, close tokyl, choice tokyl.

7.1.1. Declarers

- a)* declarer: VICTAL MODE declarer.
- b) VICTAL MODE declarer:
VICTAL MODE declarator; MODE mode indication.
- | e) VICTAL structured with FIELDS declarator:
structure tokyl, VICTAL FIELDS declarator pack.
- | fa) VICTAL MODE field TAG and MODE FOLDS continuation:

- MODE field TAG selector, comma tokyl, VICTAL MODE FOLDS continuation.
- |fb) VICTAL MODE field TAG continuation:
 - MODE field TAG selector.
- |fc) VICTAL MODE field TAG and RMODE FOLDS continuation:
 - MODE field TAG selector, comma tokyl, VICTAL RMODE FOLDS declarator.
- g)* field declarator: VICTAL FIELD declarator.
- | h) VICTAL STCWED FOLDS declarator: VICTAL STOWED declarer, VICTAL STOWED FOLDS continuation.
- i)* field selector: FIELD selector.
- j) MODE field TAG selector: TAG.
- | k) VICTAL NCNSTOWED FOLDS declarator: virtual NCNSTOWED declarer, VICTAL NCNSTOWED FOLDS continuation.
- | l) VIRACT reference to MODE declarator:
 - reference to tokyl, virtual MODE declarer.
- | m) formal reference to STOWED declarator:
 - reference to tokyl, formal STOWED declarer.
- | n) formal reference to NONSTOWED declarator:
 - reference to tokyl, virtual NONSTOWED declarer.
- | o) VICTAL ROWS of STOWED declarator:
 - VICTAL ROWS rower BRACKET, VICTAL STOWED declarer.
- | p) VICTAL ROWS of NONSTOWED declarator: VICTAL ROWS rower BRACKET, virtual NONSTOWED declarer.
- | q) VICTAL row ROWS rower:
 - VICTAL row rower, comma tokyl, VICTAL ROWS rower.
- |ra) VICTAL row rower: VICTAL lower bound, up to tokyl, VICTAL upper bound.
- |rb) VIRMAL row rower: EMPTY.
- s) virtual ICWPER bound: EMPTY.
- | t) actual LOWPER bound:
 - strict ICWPER bound, flexible tokyl option.
- | u) strict ICWPER bound: strong integral unit.
- | v) formal ICWPER bound:
 - flexible tckyl option; either tokyl.
- | w) VICTAL PROCEDURE declarator:
 - procedure tokyl, virtual PROCEDURE plan.
- x) virtual procedure with PARAMETERS MOID plan:
 - virtual PARAMETERS pack, virtual MOID declarer.
- y) virtual MODE parameter: virtual MODE declarer.
- | z) virtual void declarer: void tokyl.
- aa) virtual procedure MOID plan: virtual MCID declarer.
- bb)* parameters pack: VICTAL PARAMETERS pack.
- |cc) VICTAL union of LMOODS MOOD mode declarator:
 - union of tokyl, LMOODS MOOD and open box pack.
- dd) LOSETY LMOOD open BOX: LOSETY closed LMOOD end BOX.
- ee) LOSETY closed LMOODSETY LMOOD end BOX:
 - LOSETY closed LMOODSETY LMOOD LMOOD end BOX;
 - LOSETY open LMOODSETY LMOOD BOX.
- ff) LOSETY closed LMOODSETY LMOOD end LMOOD BOX:
 - LOSETY closed LMOODSETY LMOOD LMOOD end BOX.
- |gg) open LMOODS LMOOD BOX: LMOODS LMOOD BOX;

- open LMOODS box, comma tokyl, LMOOD ECX.
- hh) open LMCCD box: LMOOD box.
- iii) LMOODS MCCD and box:
 - union of LMOODS MOOD mode mode indication;
 - union of tokyl, open LMOODS MOOD and box pack.
- jj) MOOD and box: virtual MOOD declarer.

7.2.1. Mode Declarations

- | a) mode declaration: mode tokyl, mode definition list;
 - union of tokyl, union mode definition list;
 - structure tckyl, structured mode definition list.
- | b) mode definition: MODE mode indication, equals tokyl, actual MCDE declarer.
- | c) unicon mode definition:
 - union of LMOODS MOOD mode mode indication, equals tokyl, LMOODS MOOD and open box pack.
- | d) structured mode definition:
 - structured with FIELDS mode indication, equals tokyl, actual FIELDS declarator pack.

7.3.1. Priority Declarations

- |aa) priority declaration:
 - priority tckyl, priority definition list.
- |ab) priority definition: priority NUMBER dyadic indication, equals tokyl, NUMBER token.
- | b) one token: digit one tokyl.
- | c) TWO token: digit two tokyl.
- | d) THREE token: digit three tokyl.
- | e) FOUR token: digit four tokyl.
- | f) FIVE token: digit five tokyl.
- | g) SIX token: digit six tokyl.
- | h) SEVEN token: digit seven tokyl.
- | i) EIGHT token: digit eight tckyl.
- | i) NINE token: digit nine tokyl.

7.4.1. Identity Declarations

- |aa) identity declaration:
 - formal MCDE declarer, formal MODE definition list;
 - heap tokyl option, actual MODE declarer, actual MODE definition list;
 - procedure tokyl, procedure PARAMETY MOID definition list.
- |ab) formal MODE definition: MODE mode identifier, equals tokyl, actual MODE parameter.
- |ac) actual MODE definition: reference to MCDE mode identifier, MODE initialization option.
- |ad) MODE initialization: becomes tckyl, MODE source.
- |ae) procedure PARAMETY MOID definition:
 - procedure PARAMETY MOID mode identifier, equals

- tokyl, procedure PARAMETY MOID routine denotation.
- b) actual MODE parameter:
strong MCDE unit; MCDE transformat.

7.5.1. Operation Declarations

- | a) operation declaration:
operation tokyl, operator definition list;
operation tckyl, virtual PRAM plan, PRAM ADIC
operator definition list.
- | b) operator definition:
virtual PRAM plan, PRAM ADIC operator, equals
tokyl, actual PRAM parameter;
PRAM ADIC operator, equals tokyl, PRAM routine
denotation.
- | c) PRAM ADIC operator definition: PRAM ADIC operator,
equals tokyl, actual PRAM parameter.

8.1.1. Unitary Clauses

- | a) SORT unitary MOID clause: SORT MOID tertiary;
SORT repetitive MOID clause;
SORT MCID routine denotation;
SORT MCID confrontation.
- | b) SORT MOID tertiary: SORT MOID secondary;
SORT MOID PRIORITIES ADIC formula.
- | c) SORT MOID secondary: SORT MOID primary;
SORT MCID cohesion.
- | d) SORT MOID primary: SORT MOID base;
SORT CLOSED MOID clause.

8.1.2. Repetitive Clauses

- | a) strong repetitive void clause:
do initialization part, do loop part.
- | b) do initialization part: strong from integral clause
option, strong by integral clause option, strong to
integral clause option.
- | c) do loop part: control identifier option, strong while
boolean clause option, strong do void clause.
- | d) strong FROBYT integral clause:
FROBYT tokyl, strong serial integral clause.
- | e) control identifier:
for tokyl, integral mode identifier.
- | f) strong while boolean clause:
while tokyl, strong serial boolean clause.
- | g) strong do void clause:
do tokyl, strong unitary void clause.

8.2.0.1. Coercends

- a)* coercent: SORT COERCEND; SORTly ADAPTED to COERCEND.
- b)* SORT coercent: SORT COERCEND.
- c)* ADAPTED coercent: SORTly ADAPTED to COERCEND.
- d) strong COERCEND:
COERCEND; strongly ADAPTED to COERCEND.
- e) firm COERCEND: COERCEND; firmly ADJUSTED to COERCEND.
- |fa) meek COERCEND: COERCEND; firmly FITTED to COERCEND.
- fb) weak COERCEND: COERCEND; weakly FITTED to COERCEND.
- g) soft COERCEND:
COERCEND; softly deprocedured to COERCEND.

8.2.1.1. Dereferenced Coercends

- a) STIRMy dereferenced to MODE FORM:
reference to MODE FORM;
STIRMy FITTED to reference to MODE FORM.
- b) weakly dereferenced to reference to MODE FORM:
reference to reference to MODE FORM;
weakly FITTED to reference to reference to MODE FORM.

8.2.2.1. Deprocedured Coercends

- |aa) STIRMy deprocedured to MOID FORESE:
procedure MOID FORESE;
STIRMy FITTED to procedure MOID FORESE.
- |ab) STIRMy deprocedured to MODE confrontation:
procedure MODE confrontation;
STIRMy FITTED to procedure MODE confrontation.
- b) weakly deprocedured to MODE FORM: procedure MODE FORM;
firmly FITTED to procedure MODE FORM.
- c) softly deprocedured to MODE FORM: procedure MODE FORM;
softly deprocedured to procedure MODE FORM.

8.2.3.1. Procedured Coercends

- a) STIRMy procedured to procedure MOID FORM: MOID FORM;
STIRMy dereferenced to MOID FORM;
STIRMy procedured to MOID FORM;
STIRMy united to MOID FORM;
STIRMy widened to MOID FORM;
STIRMy rowed to MOID FORM.

8.2.4.1. United Coercends

- a) STIRMy united to union of LMOODS MOOD mode FORM:
cne cut of LMOODS MOOD mode FORM;
scme of LMOODS MOOD and but not FORM.
- b) one out of LMOODSETY MCOD RMOODSETY mode FORM:

- MOOD FORM;
firmly FITTED to MOCD FORM;
firmly procedured to MOCD FORM.
- c) some of LMOCDSETY MOOD and RMOODSETY but not LCSETY FORM:
some of LMOODSEIY and MOOD RMOODSETY but not LOSETY FCRM;
some of LMOODSETY RMOODSETY but not MOOD and LOSETY FORM.
- d) some of and LMOCD MCOD RMOODSETY but not LMCOT LCSETY FORM:
union of LMOCD MOOD RMOODSETY mode FCRM;
firmly FITTED to union of LMOOD MOOD RMOODSETY mode FORM.

8.2.5.1. Widened Coercends

- | a) strongly widened to SHONGSETY real FORM:
SHCNGSETY integral FORM;
strongly FITTED to SHONGSETY integral FCRM.
- b) strongly widened to structured with REAL field letter r letter e and REAL field letter i letter m FORM:
REAL FORM; strongly FITTED to REAL FORM;
strongly widened to REAL FORM.
- c) strongly widened to row of boolean FORM: BITS FORM;
strongly FITTED to BITS FORM.
- d) strongly widened to row of character FORM: BYTES FORM;
strongly FITTED to BYTES FORM.

8.2.6.1. Rowed Coercends

- |aa) strongly rowed to REFETY row ROWS of MODE FORM:
REFETY ROWS of MODE FORM;
strongly ADJUSTED to REFETY ROWS of MODE FORM;
strongly widened to REFETY ROWS of MODE FORM;
strongly rowed to REFETY ROWS of MODE FORM.
- |ab) strongly rowed to REFETY row of MODE FORM:
REFETY MODE FORM;
strongly ADJUSTED to REFETY MODE FORM;
strongly widened to REFETY MODE FORM;
strongly rowed to REFETY MODE FORM.

8.2.7.1. Hipped Coercends

- | a) strongly hipped to MOID base:
MCID skip; MOID jump; MOID nihil; MOID vacuum.
- | b) MOID skip: skip tokyl.
- | c) MOID jump: go to tokyl option; label identifier.
- | d) reference to MODE nihil: nil tokyl.
- | e) ROWS of MODE vacuum: open tokyl, close tokyl.

8.2.8.1. Voided Coercends

- a) strongly voided to void confrontation:
MODE confrontation.
- b) strongly voided to void FORESE: NONPROC FORESE;
strongly deprocured to NONPROC FORESE.

8.3.0.1. Confrontations

- | a) MOID confrontation: MCID assignation;
MOID identity relation; MOID cast.

8.3.1.1. Assignations

- | a) reference to MODE assignation: reference to MCDE
destination, becomes tokyl, MODE source.
- b) reference to MODE destination:
scft reference to MODE tertiary.
- c) MODE source: strong MCDE unit.

8.3.3.1. Identity Relations

- a) boclean identity relation:
scft reference to MODE tertiary, identity relator,
strong reference to MODE tertiary;
strong reference to MODE tertiary, identity
relator, scft reference to MODE tertiary.
- | b) identity relator: is tokyl; is not tokyl.

8.3.4.1. Casts

- | a) MOID cast: virtual MOID declarer, cast of tokyl,
strong MCID unit.

8.4.1. Formulas

- | a)* SORT formula: SORT MOID PRIORITIES ADIC formula.
- | b) MOID PRICRITY dyadic formula:
LMCDE PRICRITY operand, procedure with LMODE
parameter and RMODE parameter MOID PRIORITY dyadic
operator, RMCDE PRICRITY plus one operand.
- | c)* operand: MCDE PRIORITIES operand.
- | d) MODE PRICRITY operand: firm MODE PRIORITY dyadic
formula; MODE PRIORITY plus one operand.
- | e) MODE priority NINE plus one operand:
firm MCDE priority NINE plus one monadic formula;
firm MCDE secondary.
- | g) MOID priority NINE plus one monadic formula:
procedure with RMODE parameter MOID priority NINE
plus one monadic operator, RMODE priority NINE plus
one operand.

- | h)* dyadic formula: MOID PRIORITY dyadic formula.
- | i)* monadic formula:
MCID priority NINE plus one monadic formula.

8.5.0.1. Cohesions

- a) MODE cohesicn: MODE generator; MODE selection.

8.5.1.1. Generators

- a) MODE generator:
MODE local generator; MODE global generator.
- | b) reference to MODE local generator:
local tokyl, actual MODE declarer.
- | c) reference to MODE global generator:
heap tokyl, actual MODE declarer.

8.5.2.1. Selections

- | a) REFETY MODE selection: MODE field TAG selector, of
tokyl, weak REFETY structured with IFIELDSETY MODE
field TAG RFIELDSETY secondary.

8.6.0.1. Bases

- | a) MOID base: MCID mode identifier; MOID denotation;
MOID slice; MOID call.

8.6.1.1. Slices

- | aa) REFETY ROWS of MODE slice:
weak REFETY ROWSETY ROWS of MODE primary, ROWSETY
ROWS leaving ROWS indexer BRACKET.
- | ab) REFETY MODE slice: weak REFETY ROWS of MODE primary,
ROWS leaving EMPTY indexer BRACKET.
- | b) row ROWS leaving row ROWSETY indexer:
trimmer option, comma tokyl, ROWS leaving ROWSETY
indexer;
subscript, comma tokyl, ROWS leaving row ROWSETY
indexer.
- | c) row RCWS leaving EMPTY indexer:
subscript, comma tokyl, ROWS leaving EMPTY indexer.
- | d) row leaving row indexer: trimmer option.
- | e) row leaving EMPTY indexer: subscript.
- | f) trimmer: strict lower bound option, up to tokyl, stict
upper bound option, new lower bound part option;
new lower bound part.
- | g) new lower bound part: at tokyl, new lower bound.
- | h) new lower bound: strong integral unit.
- | i) subscript: strong integral unit.
- | j)* trimsript: trimmer; subscript.
- | k)* indexer: RCWS leaving ROWSETY indexer.

1)* boundscript: strict LOWER bound; new lower bound;
subscript.

8.2.6.1. Calls

a) MOID call: firm procedure with PARAMETEFS MOID
primary, actual PARAMETERS pack.

APPENDIX B

A CONTEXT-FREE SYNTAX

The context-free syntax given in this Appendix describes a superset of the modified syntax of ALGOL 68 given in Appendix A. It is a superset in the sense that the transformation from the two-level syntax described in the Report to context-free syntax greatly reduces the amount of information carried by the syntax. For example, neither modes nor coercions are expressed in the context-free syntax. Intrinsic ambiguities arising in this syntax are discussed in Chapter IV of this thesis.

Note that productions with "empty" occurring on the right-hand side have been permitted in the following syntax.

The syntactic entities are given, as far as possible, mnemonic representations recalling the notions of the syntax given in Appendix A. Prefixes, infixes, and suffixes used in the composition of the syntactic entities are listed below together with their meanings in terms of the syntax given in Appendix A. This table should be regarded only as an aid for reading the syntax.

ac.....actual	mo.....mould
align....alignment	mon.....monadic
assig....assignation	nos.....NONSTOWED
beg.....begin	ong.....long
bool.....boolean	op.....operation
char.....character	p.....priority
choi.....choice	par.....parameter
cl.....clause	pat.....pattern
clo.....closed	pk.....pack
col.....collateral	prel.....prelude
collect..collection	pri.....priority
cond.....conditional	proc.....procedure
conf.....conformity case	pt.....point
cnfrcnt.cnfrntation	ref.....reference to
cos.....CONFASE	rel.....relation
dec.....declaration	replic....replication
decer....declarer	replit....replicated literal
den.....denotation	rtn.....routine
dig.....digit	s.....sequence, list
dor.....declarator	ser.....serial
dy.....dyadic	sho.....short
ety.....empty	sp.....sequence proper
exp.....exponent	spec.....specification
flo.....floating	sta.....statement
fo.....formal	stag.....stagnant

form.....formula	stan.....standard
id.....identity	sto.....STOWED
iden.....identifier	str.....structured with
ind.....indication	sui.....suite
init.....initialization	suppress..suppressible
insert...insertion	sy.....symbol
int.....integer	uni.....unitary
lab.....label	unio.....union of
let.....letter	vi.....virtual
lob.....lower bound	vo.....void

Terminal productions of this syntax are "letter", "character", "indicant", "dyindicant", "monindicant", and all entities ending with "*sy". For their representations, see Appendix C.

```

program: clccl.
labsety: empty;
        labsety, lab.
empty: .
lab: iden, lat*sy.
iden: letter;
      iden, letter;
      iden, digit.
digit: dig0*sy;
       dignot0.
dignot0: dig1*sy;
         dig2*sy;
         dig3*sy;
         dig4*sy;
         dig5*sy;
         dig6*sy;
         dig7*sy;
         dig8*sy;
         dig9*sy.

clause: clocl;
```



```

colcl:
  colcl;
  coscl.

cloc1: begin*sy, sercl, end*sy;
       open*sy, sercl, close*sy.

colcl: begin*sy, balance, end*sy;
       open*sy, balance, close*sy;
       parallel*sy, begin*sy, balance, end*sy;
       parallel*sy, open*sy, balance, close*sy.
balance: unit, comma*sy, unit;
        balance, comma*sy, unit.

coscl: condbeg*sy, sercl, condchoicl, condend*sy;
       casebeg*sy, sercl, casechoicl, casend*sy;
       confbeg*sy, sercl, confchoicl, confend*sy;
       cosbeg*sy, sercl, choicl, cosend*sy.
condchoicl: stropthef*sy, sercl, condchoicl;
            condincl, condoutclety.
casechoicl: caseincl, caseoutclety.
confchoicl: cnfincl, confoutclety.
choicl: choicl1;
        choicl2;
        choicl3.
choicl1: briefthef*sy, sercl, choicl1;
        incl1, outclety;
        incl1, outcl1.
choicl2: incl2, outclety;
        incl2, outcl2.
choicl3: incl3, outclety;
        incl3, outcl3.
condincl: condin*sy, sercl.
caseincl: casein*sy, balance.
cnfincl: cnfin*sy, confunit;
        cnfin*sy, cnfbalance.
incl1: in*sy, sercl.
incl2: in*sy, balance.
incl3: in*sy, confunit;
        in*sy, cnfbalance.
condoutclety: empty;
              condout*sy, sercl;
              condoutbeg*sy, sercl, condchoicl.
caseoutclety: empty;
              caseout*sy, sercl;
              caseoutbeg*sy, sercl, casechoicl.
confoutclety: empty;
              confout*sy, sercl;
              confoutbeg*sy, sercl, confchoicl.
outclety: empty;
          out*sy, sercl.
outcl1: outbeg*sy, sercl, choicl1.
outcl2: outbeg*sy, sercl, choicl2.

```



```

outcl3: outreg*sy, sercl, choicl3.
confbalance: confunit, comma*sy, confunit;
             confbalance, comma*sy, confunit.
confunit: spec, unit.
spec: open*sy, fodecer, iden, close*sy, choice*sy;
      open*sy, fodecer, close*sy, choice*sy.

sercl: suitrains;
      decprels, suitrains.
decprels: decprel;
          decprels, decprel.
decprel: decs, goon*sy;
          staprels, decs, goon*sy.
staprels: unit, goon*sy;
          staprels, unit, goon*sy.
suitrains: train;
           suitrains, completer, train.
train: labsety, unit;
       train, goon*sy, labsety, unit.
completer: completion*sy, lab.

unit: tertiary;
      rtnden;
      repetitivecl;
      confront.
tertiary: secondary;
          formula.
secondary: primary;
           cohesion.
primary: clause;
         base.

rtnden: foparpk, vovidecer, rtn*sy, unit;
        vovidecer, rtn*sy, unit.
foparpk: open*sy, fopars, clcse*sy.
fopars: fopar;
        fopars, comma*sy, fopar.
fopar: fodecer, iden;
       fopar, comma*sy, iden.
vovidecer: videcer;
          void*sy.

repetitivecl: fromclety, byclety, toclety, foridenety,
              whileclety, docl.
fromclety: empty;
           from*sy, sercl.
byclety: empty;
         by*sy, sercl.
toclety: empty;
         to*sy, sercl.
foridenety: empty;
            for*sy, iden.

```



```

whileclety: empty;
            while*sy, sercl.
docl: do*sy, unit.

ccnfrcnt: assig;
            idrel;
            cast.
assig: tertiary, becomes*sy, unit.
idrel: tertiary, is*sy, tertiary;
            tertiary, isnot*sy, tertiary.
cast: vovidecer, cast*sy, unit.

formula: monform;
            dyform.
dyform: p1form;
            p2form;
            p3form;
            p4form;
            p5form;
            p6form;
            p7form;
            p8form;
            p9form.
p1form: p1operand, p1operator, p2operand.
p2form: p2operand, p2operator, p3operand.
p3form: p3operand, p3operator, p4operand.
p4form: p4operand, p4operator, p5operand.
p5form: p5operand, p5operator, p6operand.
p6form: p6operand, p6operator, p7operand.
p7form: p7operand, p7operator, p8operand.
p8form: p8operand, p8operator, p9operand.
p9form: p9operand, p9operator, p10operand.
p1operand: p1form;
            p2operand.
p2operand: p2form;
            p3operand.
p3operand: p3form;
            p4operand.
p4operand: p4form;
            p5operand.
p5operand: p5form;
            p6operand.
p6operand: p6form;
            p7operand.
p7operand: p7form;
            p8operand.
p8operand: p8form;
            p9operand.
p9operand: p9form;
            p10operand.
p10operand: monform;
            secondary.

```



```

mcnform: monind, p10operand.
p1operator: dyind.
p2operator: dyind.
p3operator: dyind.
p4operator: dyind.
p5operator: dyind.
p6operator: dyind.
p7operator: dyind.
p8operator: dyind.
p9operator: dyind.
monind: shongsety, monindicant.
dyind: shongsety, dyindicant.

cohesion: generator;
          selection.
generator: local*sy, acdecerc;
          heap*sy, acdecerc.
selection: iden, of*sy, secondary.

base: iden;
      den;
      jump;
      skip*sy;
      nil*sy;
      slice;
      call;
      vacuum.

den: shongsety, intden;
     shongsety, realden;
     boolden;
     charden;
     shongsety, bitsden;
     stringden;
     formatden.
shongsety: empty;
           longs;
           shorts.
longs: long*sy;
       longs, long*sy.
shorts: short*sy;
        shorts, short*sy.

intden: digit;
        intden, digit.

realden: intden, pt*sy, intden, expety;
         pt*sy, intden, expety;
         intden, exp.
expety: empty;
        exp.
exp: letE*sy, flusminusety, intden.

```



```

plusminusety: empty;
               plus*sy;
               minus*sy.

boolden: true*sy;
         false*sy.

charden: briefqucte*sy, stringitem, briefquote*sy;
         stropquote*sy, stringitem, stropquote*sy.
stringitem: briefquote*sy, briefquote*sy;
            stropquote*sy, stropquote*sy;
            character.

bitsden: dig2*sy, letR*sy, radix2bytes;
         dig4*sy, letR*sy, radix4bytes;
         dig8*sy, letR*sy, radix8bytes;
         dig1*sy, dig6*sy, letR*sy, radix16bytes.
radix2bytes: radix2byte;
            radix2bytes, radix2byte.
radix4bytes: radix4byte;
            radix4bytes, radix4byte.
radix8bytes: radix8byte;
            radix8bytes, radix8byte.
radix16bytes: radix16byte;
            radix16bytes, radix16byte.
radix2byte: dig0*sy;
            dig1*sy.
radix4byte: radix2byte;
            dig2*sy;
            dig3*sy.
radix8byte: radix4byte;
            dig4*sy;
            dig5*sy;
            dig6*sy;
            dig7*sy.
radix16byte: radix8byte;
            dig8*sy;
            dig9*sy;
            letA*sy;
            letB*sy;
            letC*sy;
            letD*sy;
            letE*sy;
            letF*sy.

stringden: briefquote*sy, briefquote*sy;
           stropquote*sy, stropquote*sy;
           briefquote*sy, stringitemsp, briefquote*sy;
           stropquote*sy, stringitemsp, stropquote*sy.
stringitemsp: stringitem, stringitem;
             stringitemsp, stringitem.

```



```

formatden: formatter*sy, collects, formatter*sy.
collects: collect;
          collects, comma*sy, collect.
collect: picture;
          insertety, replicety, open*sy, collects, close*sy,
          insertety.
picture: patternety, insertety.
patternety: empty;
            intpat;
            realpat;
            boclpat;
            charpat;
            stringpat;
            ccomplexpat;
            bitspat.
insertety: literalety, insertsety.
insertsety: empty;
            insertsety, insert.
insert: replicety, align, literalety.
replicety: empty;
            replic.
replic: intden;
        letN*sy, clause.
align: letK*sy;
        letX*sy;
        letY*sy;
        letI*sy;
        letP*sy.
literateley: empty;
            literal.
literal: replicety, stringcharden, replitsety.
replitsety: empty;
            replitsety, replic, stringcharden.
stringcharden: charden;
              stringden.
signmoety: empty;
            insertety, signframe;
            insertety, replicety, zeroframe, signframe.
zeroframe: letZ*sy.
signframe: plus*sy;
            minus*sy.

intpat: signmoety, intmo;
        insertety, letC*sy, open*sy, literals, close*sy.
intmc: insertety, replicety, suppressety, digframe;
        intmo, insertety, replicety, suppressety, digframe.
digframe: zercframe;
          letD*sy.
suppressety: empty;
            letS*sy.
literals: literal;
          literals, comma*sy, literal.

```



```

realpat: signmoety, realm;
        floptmo.
realm: intmo, insertety, suppressety, ptframe, intmoety;
        insertety, suppressety, ptframe, intmc.
intmoety: empty;
        intro.
ptframe: pt*sy.
floptmo: stagmo, insertety, suppressety, expframe,
        signmoety, intmo.
stagmo: signmoety, intmo;
        signmoety, realm.
expframe: letE*sy.

boolpat: insertety, letB*sy, boolmoety.
bclmcety: empty;
        open*sy, literal, comma*sy, literal, close*sy.

charpat: insertety, suppressety, charframe.
charframe: letA*sy.

complexpat: realpat, insertety, suppressety, complexframe,
        realpat.
complexframe: letI*sy.

stringpat: insertety, stringframe;
        charframesp;
        insertety, replic, suppressety, charframe.
stringframe: letT*sy.
charframesp: insertety, replicety, suppressety, charframe,
        insertety, replicety, suppressety, charframe;
        charframesp, insertety, replicety, suppressety,
        charframe.

bitspat: radixmo, intmo.
radixmo: insertety, radix, letR*sy.
radix: dig2*sy;
        dig4*sy;
        dig8*sy;
        dig1*sy, dig6*sy.

jump: goto*sy, iden.

slice: primary, sub1*sy, indexers, bus1*sy;
        primary, sub2*sy, indexers, bus2*sy;
        primary, open*sy, indexers, close*sy.
indexers: trimscriptety;
        indexers, comma*sy, trimscriptety.
trimscriptety: subscript;
        trimmerety.
subscript: unit.
trimmerety: boundety, upto*sy, boundety, newlobety;

```



```

        newlobety.
boundety: empty;
        unit.
newlobety: empty;
        at*sy, unit.

call: primary, open*sy, acpars, close*sy.
acpars: acpar;
        acpars, comma*sy, acpar.
acpar: unit.

vacuum: open*sy, close*sy.

decs: unidec;
        decs, comma*sy, unidec.

unidec: modedec;
        strdec;
        uniodec;
        pridec;
        iddec;
        opdec.

modedec: mode*sy, modeind, equal*sy, acdec;
        modedec, comma*sy, modeind, equals*sy, acdec.
strdec: str*sy, modeind, equals*sy, acfieldpk;
        strdec, comma*sy, modeind, equals*sy, acfieldpk.
acfieldpk: open*sy, acfieldors, close*sy.
uniodec: union*sy, modeind, equals*sy, videcerspk;
        uniodec, comma*sy, modeind, equals*sy, videcerspk.
videcerspk: open*sy, videcersp, close*sy.
videcersp: videcer, comma*sy, videcer;
        videcersp, comma*sy, videcer.

pridec: pri*sy, dyind, equals*sy, dignot0;
        pridec, comma*sy, dyind, equals*sy, dignot0.

iddec: iddec1s;
        iddec2s;
        iddec3s.
iddec1s: fodecer, idacpar;
        iddec1s, comma*sy, idacpar.
idacpar: iden, equals*sy, acpar.
iddec2s: heapety, acdec, ideninit;
        iddec2s, comma*sy, ideninit.
heapety: empty;
        heap*sy.
ideninit: iden;
        iden, becomes*sy, unit.
iddec3s: proc*sy, idrtnden;
        iddec3s, comma*sy, idrtnden.
idrtnden: iden, equals*sy, rtnden.

```



```

opdec: opdec1s;
      opdec2s.
opdec1s: op*sy, opacpar1;
         opdec1s, comma*sy, opacpar1.
opacpar1: viplan, adicind, equals*sy, unit;
          adicind, equals*sy, rtnden.
opdec2s: op*sy, viplan, opacpar2;
         opdec2s, comma*sy, opacpar2.
opacpar2: adicind, equals*sy, unit.
viplan: open*sy, videcer, close*sy, vovidecer;
        open*sy, videcer, comma*sy, videcer, close*sy,
        vovidecer.
adicind: monind;
        dyind.

acdecer: acnosdor;
        acstodor;
        modeind.
acncsdor: procdor;
        uniodor;
        acrefdor.
acrefdor: ref*sy, videcer.
acstodor: acstrdor;
        acrowdor.
acstrdor: str*sy, acfieldpk.
acfieldors: acfieldor;
            acfieldors, comma*sy, acfieldor.
acfieldor: acstodor, iden;
            vincsdor, iden;
            modeind, iden;
            acfieldor, comma*sy, iden.
acrowdor: sub1*sy, acrowers, bus1*sy, acstodor;
          sub2*sy, acrowers, bus2*sy, acstodor;
          open*sy, acrowers, close*sy, acstodor;
          sub1*sy, acrowers, bus1*sy, vinosdor;
          sub2*sy, acrowers, bus2*sy, vinosdor;
          open*sy, acrowers, close*sy, vinosdor;
          sub1*sy, acrowers, bus1*sy, modeind;
          sub2*sy, acrowers, bus2*sy, modeind;
          open*sy, acrowers, close*sy, modeind.
acrowers: acrower;
          acrowers, comma*sy, acrower.
acrower: acbound, upto*sy, acbound.
acbound: unit;
        unit, flex*sy.

fodecer: fonosdor;
        fostodor;
        modeind.
foncsdor: procdor;
        uniodor;

```



```

        forefdor.
forefdor: ref*sy, fostodor;
        ref*sy, vincsdor.
fostodor: fostrdor;
        forcwdor.
fostrdor: str*sy, open*sy, fofieldors, close*sy.
fofieldors: fofieldor;
        fofieldors, comma*sy, fofieldor.
fofieldor: fostodor, iden;
        vincsdor, iden;
        modeind, iden;
        fofieldor, comma*sy, iden.
forowdor: sub1*sy, forowersety, bus1*sy, fostodor;
        sub2*sy, forowersety, bus2*sy, fostodor;
        open*sy, forowersety, close*sy, fostodor;
        sub1*sy, forowersety, bus1*sy, vincsdor;
        sub2*sy, forowersety, bus2*sy, vinosdor;
        open*sy, forowersety, close*sy, vinosdor;
        sub1*sy, forowersety, bus1*sy, modeind;
        sub2*sy, forowersety, bus2*sy, modeind;
        open*sy, forowersety, close*sy, modeind.
forowersety: forcwerety;
        forowersety, comma*sy, forowerety.
forowerety: empty;
        foboundety, upto*sy, foboundety.
foboundety: empty;
        flex*sy;
        either*sy.

videcer: vinosdor;
        vistodor;
        modeind.
vincsdor: procdor;
        uniodor;
        virefdor.
virefdor: ref*sy, videcer.
vistodor: vistrdor;
        virowdor.
vistrdor: str*sy, open*sy, vifieldors, close*sy.
vifieldors: vifieldor;
        vifieldors, comma*sy, vifieldor.
vifieldor: videcer, iden;
        vifieldor, comma*sy, iden.
virowdor: sub1*sy, virowersety, bus1*sy, videcer;
        sub2*sy, virowersety, bus2*sy, videcer;
        open*sy, virowersety, close*sy, videcer.
virowersety: virowerety;
        virowersety, comma*sy, virowerety.
virowerety: empty;
        upto*sy.

procdor: proc*sy, vovidecer;

```



```
        proc*sy, open*sy, videcer, close*sy, vovidecer;  
        proc*sy, videcerspk, vovidecer.  
uniodor: unio*sy, videcerspk.  
modeind: modestan;  
        indicant.  
modestan: shongsety, int*sy;  
        shongsety, real*sy;  
        bool*sy;  
        char*sy;  
        format*sy;  
        shongsety, hits*sy;  
        shongsety, bytes*sy;  
        string*sy;  
        sema*sy;  
        shongsety, complex*sy;  
        file*sy.
```


APPENDIX C

A REPRESENTATION LANGUAGE

This representation language for ALGOL 68 was designed for the one-pass implementation proposed in this thesis. The symbols for which representations are given correspond to terminals of the syntax presented in Appendix A. Where such a symbol is mentioned, the corresponding terminal from the context-free syntax given in Appendix B, if one exists, follows in parentheses.

The "stropping" convention (method of representing bold-faced or underlined symbols) chosen is that of a "bold-face shift", that is, an apostrophe ('), followed by a sequence of letters (upper or lower case) or digits, ending on a "light-face shift", that is, any mark different from the representation of a letter or digit. A symbol formed in this fashion is called a "strop-word". Note that 'IF is a different strop-word than 'if.

The representations are grouped in an order similar to that used in Chapter 3 of the Report.

1. Letter Tokens

The production for the metanotion "ALPHA" is altered [R1.1.4] as follows;

ALPHA: a; b; ... ; z; lower case a; lower case b; ... ;
lower case z.

Then the following are representations for letter symbols;

letter a symbol (letA*sy)	A
letter lower case a symbol	a
letter b symbol (letB*sy)	B
letter lower case b symbol	b
...	
letter z symbol (letZ*sy)	Z
letter lower case z symbol	z

The terminal production "letter" of the context-free syntax includes all letter-tokens.

2. Denotation Tokens

Digit Tokens.

digit zero symbol (dig0*sy)	0
digit one symbol (dig1*sy)	1
...	
digit nine symbol (dig9*sy)	9

Following are other denotation-tokens.

point symbol (pt*sy)	.
plus symbol (plus*sy)	+
minus symbol (minus*sy)	-
true symbol (true*sy)	'TRUE
false symbol (false*sy)	'FALSE
formatter symbol (formatter*sy)	\$

No representation has been provided for the "times-ten-to-the-power-symbol" (this was not included in the context-free syntax).

3. Character Tokens

These tokens are used in string- and character-denotations and in comments. Representations for all of them except the "space-symbol" are given elsewhere in this Appendix. Note, in Appendix A, that "plus-i-times-symbol" has been omitted from, and that plus-symbol and minus-symbol have been added to, the list of character-tokens as given in the Report. The space-symbol is represented by a blank, which will be significant as it will appear in string- or character- denotations or comments. Insignificant blanks within the source text may appear anywhere that comments may; that is, between but not within symbols.

The terminal "character" of the context-free syntax corresponds to the above character-tokens plus all terminal productions of "other-string-item" and "other-comment-item". Productions are added [R1.1.5.c] for other-comment-item and other-string-item so that their terminal productions include most, if not all, available graphic symbols of the physical machine, apart from those already included as character-tokens. Note that other-comment-item does not include "comment-symbol" and other-string-item does not include quote-symbol among their respective terminal productions [R1.1.5.c].


```

strop conformity case begin symbol
    (confbeg*sy)          'CASEC
brief thef symbol (briefthef*sy) | : !:
strop thef symbol (stropthef*sy) 'THEF
brief CCNFASE in symbol (in*sy)  | !
strop conditional in symbol
    (condin*sy)          'THEN
strop case in symbol (casein*sy)  'IN
strop conformity case in symbol
    (confin*sy)          'IN
brief CCNFASE out symbol (out*sy) | !
strop conditional out symbol
    (condout*sy)         'ELSE
strop case out symbol (caseout*sy) 'OUT
strop conformity case out symbol
    (confout*sy)         'OUT
brief CCNFASE end symbol (cosend*sy) )
strop conditional end symbol
    (condend*sy)         'FI
strop case end symbol (casend*sy)  'ESAC
strop conformity case end symbol
    (confend*sy)         'CESAC
brief CCNFASE out begin symbol
    (outbeg*sy)          | : !:
strop conditional out begin symbol
    (ccndoutbeg*sy)      'ELSF
strop case out begin symbol
    (caseoutbeg*sy)      'CUSE
strop conformity case out begin symbol
    (confoutbeg*sy)      'OUSEC
choice symbol (choice*sy)         : ..
from symbol (from*sy)             'FROM
by symbol (by*sy)                 'BY
to symbol (to*sy)                 'TO
for symbol (for*sy)               'FOR
while symbol (while*sy)           'WHILE
do symbol (do*sy)                 'DO

```

7. Sequencing Tokens

```

go on symbol (goon*sy)           ; ..
completion symbol (ccompletion*sy) 'EXIT .
go to symbol (goto*sy)           'GOTO 'GO 'GO 'TO

```

8. Hip Tokens

```

skip symbol (skip*sy)           'SKIP
nil symbol (nil*sy)             'NIL

```


9. Special Tokens

brief quote symbol (briefquote*sy)	"
strop quote symbol (stropquote*sy)	'QUOTE
brief comment symbol	# { }
strop comment symbol	'COMMENT 'CO
	'PRAGMAT 'PR

When the quote-symbol 'QUOTE commences a string- or character-denotation, then the blank following it, if there is one, is not part of the string- or character-denotation. For example, 'QUOTE A'QUOTE is the same as "A", while 'QUOTE;'QUOTE is the same as ";". This is a consequence of the chosen stropping convention.

10. Mode Standards

integral symbol (int*sy)	'INT 'INTEGER
real symbol (real*sy)	'REAL
boolean symbol (bool*sy)	'BOCL 'BOOLEAN
character symbol (char*sy)	'CHAR 'CHARACTER
format symbol (format*sy)	'FORMAT
bits symbol (bits*sy)	'BITS
bytes symbol (bytes*sy)	'BYTES
string symbol (string*sy)	'STRING
sema symbol (sema*sy)	'SEMA
complex symbol (complex*sy)	'COMPL 'CCOMPLEX
file symbol (file*sy)	'FILE

The alternatives given are defined by inserting declarations such as

```
'MODE 'INTEGER = 'INT
```

into the standard-prelude.

11. Other Terminals

Further terminals in the syntax are indicant, dyadic-indicant, and monadic-indicant. In the context-free syntax

these are: "indicant", "dyindicant", and "monindicant" respectively. Productions for these may be added [R1.1.5.c], and this is done as follows;

indicant: strop word.

dyadic indicant: strop word;

special character dyadic indicant.

monadic indicant: strop word;

special character monadic indicant.

Special characters are those characters other than letters and digits.

No strop-word may be a terminal production of both indicant and monadic-indicant [R1.1.5.b]. No strop-word that is given in sections two through nine of this Appendix (these may be called "program-words") may be a terminal production of indicant, monadic-indicant, or dyadic-indicant. No representation comprising special characters given in sections two through nine of this Appendix (these may be called "special-symbols") may be produced by indicant, dyadic-indicant, or monadic-indicant, with the exception of +, -, and =. Whether these marks are special-symbols or not is easily distinguishable by context. These conditions and the chosen stropping convention ensure that "no sequence of representations of symbols can be confused with any other such sequence" [R3.1.2.c]. An example of an ambiguity that results without the above conditions is as follows;


```

cp to = (int a, int b) int: skip;
from i to j do skip; .

```

Note that since indicants, dyadic-indicants, and monadic-indicants are themselves symbols, no blanks or comments may be embedded within them.

The entity "special-character-dyadic-indicant" is denoted by "D" and given as follows:

```

D: D , "*" ;
   D , "/" ;
   D , ":" ;
   D , "=" ;
   T ;
   "*" ;
   "/" ;
   "=" .

```

Special characters have been quoted to distinguish them from syntactic marks. "T" has as its direct productions each member of a set of special characters not including "*", "/", ":", "=", "\$", "(", ")", ".", ",", ";", "@", "#", "", "'", "[", "]", "{", "}", "!", "|", or "ø". The entity "special-character-monadic-indicant" is denoted by "M" and given as follows:

```

M: M , "*" ;
   M , "/" ;
   M , ":" ;
   M , "=" ;
   T .

```

"T" is defined as above. Now if "T" produces "+", "-", "<", ">", "~", "&", "&" and "%" then the above rules allow representations as given below for standard "operator-tokens", whose corresponding declarations will be included

in the standard-prelude. For mnemonic reasons, the use of "AB" as a suffix for "and-becomes" is consistent. Where alternatives are given, these are introduced by declaration in the standard-prelude so there will be only one representation per indicant, though some indicants may initially have the same meaning.

minus and becomes symbol	'MINUSAB -:=
plus and becomes symbol	'PLUSAB +:=
times and becomes symbol	'TIMESAB *:=
divided by and becomes symbol	'DIVAB /:=
over and becomes symbol	'OVERAB %:=
modulo and becomes symbol	'MODAB %:=
plus and becomes symbol	'PLUSAB +:=
or symbol	'OR
and symbol	'AND &
differs from symbol	'NE != #
is less than symbol	'LT <
is at most symbol	'LE <=
is at least symbol	'GE >=
is greater than symbol	'GT >
divided by symbol	'DIV /
over symbol	'OVER %
times symbol	'TIMES *
add symbol	'PLUS +
subtract symbol	'MINUS -
modulo symbol	'MOD %:
th element of symbol	'ELEM
lower bound of symbol	'LWB
upper bound of symbol	'UPB
lower state of symbol	'LWS
upper state of symbol	'UPS
entier symbol	'ENTIER
not symbol	'NOT
down symbol	'DCWN
up symbol	'UP
integral to sema symbol	'LEVEL
sema to integral symbol	'LEVEL
plus i times symbol	'I
power symbol	'PCW **
shift left symbol	'SHL
shift right symbol	'SHR
absolute value of symbol	'ABS
binomial symbol	'BIN
representation of symbol	'REPR
lengthen symbol	'LENG
shorten symbol	'SHORTEN

odd symbol	'ODD
sign symbol	'SIGN
round symbol	'RCUND
real part of symbol	'RE
imaginary part of symbol	'IM
conjugate of symbol	'CCNJ
argument of symbol	'ARG
boolean to bits symbol	'PACK
characters to bytes symbol	'PACK
equality symbol	'EQ =

It should be noted that the above syntax for special-character-monadic- and dyadic-indicants does not permit *, =, or / as monadic-indicants, nor are they permitted to appear at the start of monadic-indicants. This is to prevent ambiguities and/or indeterminations from occurring. For example; $A**B$ could be $A ** B$ or $A * *B$ if * could be monadic; $A<=B$ could be $A <= B$ or $A < =B$ if = could be monadic; in $(/:A \dots$, the (/ may be an alternate-sub-symbol or /: may be a monadic-indicant (this is resolvable at the matching bus-symbol or close-symbol, but the indetermination is not desirable). 'TIMES, 'EQ, and 'DIV, however, are permitted as monadic-indicants. Note that $\neg=:$ was not given as an alternate representation for the is-not-symbol. This is because $A:\neg=:B$ might be $A: \neg=:B$ where $A:$ is a label and $\neg=:B$ is a monadic-formula or $A : \neg=: B$ where $: \neg=:$ is the is-not-symbol.

B30035